AD-A047 602 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO F/G 5/2
USE OF A HIERARCHICAL DATA MODEL TO SUPPORT A RELATIONAL DATA M--ETC(U)
AUG 77 E K CONOLEY

UNCLASSIFIED AFIT-CI-78-3 NL

OF
AD
A047602

END
DATE
FILMED
1 -78
DDC

CI 78-3

1

USE OF A HIERARCHICAL DATA MODEL TO SUPPORT A RELATIONAL DATA MANAGEMENT SYSTEM

APPROVED:

AC Date

Neil Dale

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CI 78-3	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Use of a Hierarchical Data Model to Support a Relational Data Management System		5. TYPE OF REPORT & PERIOD COVERED Thesis 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ellis K. Conoley		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT Student University of Texas, Austin TX		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/CI WPAFB OH 45433		12. REPORT DATE August 1977 13. NUMBER OF PAGES 80 Pages
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)* Approved for Public Release; Distribution Unlimited		
17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*		
18. SUPPLEMENTARY NOTES APPROVED FOR PUBLIC RELEASE AFR 190-17. JERRAL F. GUESS, Captain, USAF Director of Information, AFIT		
19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*		
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

INSTRUCTIONS FOR PREPARATION OF REPORT DOCUMENTATION PAGE

RESPONSIBILITY. The controlling DoD office will be responsible for completion of the Report Documentation Page, DD Form 1473, in all technical reports prepared by or for DoD organizations.

CLASSIFICATION. Since this Report Documentation Page, DD Form 1473, is used in preparing announcements, bibliographies, and data banks, it should be unclassified if possible. If a classification is required, identify the classified items on the page by the appropriate symbol.

COMPLETION GUIDE

General. Make Blocks 1, 4, 5, 6, 7, 11, 13, 15, and 16 agree with the corresponding information on the report cover. Leave Blocks 2 and 3 blank.

Block 1. Report Number. Enter the unique alphanumeric report number shown on the cover.

Block 2. Government Accession No. Leave Blank. This space is for use by the Defense Documentation Center.

Block 3. Recipient's Catalog Number. Leave blank. This space is for the use of the report recipient to assist in future retrieval of the document.

Block 4. Title and Subtitle. Enter the title in all capital letters exactly as it appears on the publication. Titles should be unclassified whenever possible. Write out the English equivalent for Greek letters and mathematical symbols in the title *(see "Abstracting Scientific and Technical Reports of Defense-sponsored RDT/E," AD-667 000)*. If the report has a subtitle, this subtitle should follow the main title, be separated by a comma or semicolon if appropriate, and be initially capitalized. If a publication has a title in a foreign language, translate the title into English and follow the English translation with the title in the original language. Make every effort to simplify the title before publication.

Block 5. Type of Report and Period Covered. Indicate here whether report is interim, final, etc., and, if applicable, inclusive dates of period covered, such as the life of a contract covered in a final contractor report.

Block 6. Performing Organization Report Number. Only numbers other than the official report number shown in Block 1, such as series numbers for in-house reports or a contractor/grantee number assigned by him, will be placed in this space. If no such numbers are used, leave this space blank.

Block 7. Author(s). Include corresponding information from the report cover. Give the name(s) of the author(s) in conventional order *(for example, John R. Doe or, if author prefers, J. Robert Doe)*. In addition, list the affiliation of an author if it differs from that of the performing organization.

Block 8. Contract or Grant Number(s). For a contractor or grantee report, enter the complete contract or grant number(s) under which the work reported was accomplished. Leave blank in in-house reports.

Block 9. Performing Organization Name and Address. For in-house reports enter the name and address, including office symbol, of the performing activity. For contractor or grantee reports enter the name and address of the contractor or grantee who prepared the report and identify the appropriate corporate division, school, laboratory, etc., of the author. List city, state, and ZIP Code.

Block 10. Program Element, Project, Task Area, and Work Unit Numbers. Enter here the number code from the applicable Department of Defense form, such as the DD Form 1498, "Research and Technology Work Unit Summary" or the DD Form 1634, "Research and Development Planning Summary," which identifies the program element, project, task area, and work unit or equivalent under which the work was authorized.

Block 11. Controlling Office Name and Address. Enter the full, official name and address, including office symbol, of the controlling office. *(Equates to funding/sponsoring agency. For definition see DoD Directive 5200.20, "Distribution Statements on Technical Documents.")*

Block 12. Report Date. Enter here the day, month, and year or month and year as shown on the cover.

Block 13. Number of Pages. Enter the total number of pages.

Block 14. Monitoring Agency Name and Address *(if different from Controlling Office)*. For use when the controlling or funding office does not directly administer a project, contract, or grant, but delegates the administrative responsibility to another organization.

Blocks 15 & 15a. Security Classification of the Report: Declassification/Downgrading Schedule of the Report. Enter in 15 the highest classification of the report. If appropriate, enter in 15a the declassification/downgrading schedule of the report, using the abbreviations for declassification/downgrading schedules listed in paragraph 4-207 of DoD 5200.1-R.

Block 16. Distribution Statement of the Report. Insert here the applicable distribution statement of the report from DoD Directive 5200.20, "Distribution Statements on Technical Documents."

Block 17. Distribution Statement *(of the abstract entered in Block 20, if different from the distribution statement of the report)*. Insert here the applicable distribution statement of the abstract from DoD Directive 5200.20, "Distribution Statements on Technical Documents."

Block 18. Supplementary Notes. Enter information not included elsewhere but useful, such as: Prepared in cooperation with . . . Translation of *(or by)* . . . Presented at conference of . . . To be published in . . .

Block 19. Key Words. Select terms or short phrases that identify the principal subjects covered in the report, and are sufficiently specific and precise to be used as index entries for cataloging, conforming to standard terminology. The DoD "Thesaurus of Engineering and Scientific Terms" *(TEST)*, AD-672 000, can be helpful.

Block 20. Abstract. The abstract should be a brief *(not to exceed 200 words)* factual summary of the most significant information contained in the report. If possible, the abstract of a classified report should be unclassified and the abstract to an unclassified report should consist of publicly- releasable information. If the report contains a significant bibliography or literature survey, mention it here. For information on preparing abstracts see "Abstracting Scientific and Technical Reports of Defense-Sponsored RDT&E," AD-667 000.

☆ U.S. GOVERNMENT PRINTING OFFICE : 1973—729-091/1431 3-1

(6) USE OF A HIERARCHICAL DATA MODEL TO SUPPORT A RELATIONAL DATA MANAGEMENT SYSTEM.

by

(9) Master's thesis,

(10) ELLIS KNOX/CONOLEY B.A.

(14) AFIT-CI-78-3

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS

(12) 85p.

THE UNIVERSITY OF TEXAS AT AUSTIN

(11) August 1977

012 200

4B

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

Each Data Base Management System (DBMS) has its own unique approach to the data management problem. Three basic management systems, hierarchical, relational and network, have evolved, each of which constructs the logical view of a data base in a different way. None of the approaches is superior to the other two, rather each has its own special advantages and applications.

Lowenthal and others have discussed the design of a general purpose DBMS kernal which is able to handle several logical views of a data base instead of only one logical view [15]. The kernel accepts many different "front ends" by having general units for parsing, data access, work areas, buffers, etc. The characteristics of such a kernel should not affect the user, i.e., the personality of the management system should not extend through the user-friendly interface.

The design of such a kernel is a complex undertaking. In order to discover what specific functions should be a part of the kernel one should study what the kernel must do for a specific model in a specific instance. The action which would be taken by each model must be noted to assure that the kernel is able to adequately process requests by a front end system. In addition, the relationships between current approaches must be considered so that any duplication of effort by a particular front end is avoided.

This thesis is an investigation of some of the relationships that exist between hierarchical data base organizations and relational data base organizations. It discusses these relationships in terms of the problems and solutions involved in implementing a relational front end to a hierarchical data base. A solution is presented which defines all relational algebra operations in terms of hierarchical operations. The system does not have a query optimizer, but is constructed with facilities for the addition of one. It provides a vehicle which can be modified to study other relational organizations in the hierarchy.

CHAPTER II consists of term definitions and a brief discussion of the relational and hierarchical models. CHAPTER III discusses the specific problem of designing a relational front end to a hierarchy, arriving at a possible organization. CHAPTER IV is a study of the algorithms necessary to implement the algebra. CHAPTER V mentions a possible extension to the algorithms of CHAPTER IV, and CHAPTER VI is a summary.

CHAPTER II

DEFINITIONS

1. Data Base

> ... a collection of interrelated data stored together without harmful or unnecessary redundancy to serve one or more applications in an optimal fashion; the data are stored so that they are independent of programs which use the data; a common and controlled approach is used in adding new data and in modifying existing data within the data base. [16]

2. Data Base Management System

A Data Base Management System (DBMS) is a software system for managing data bases on a computer. Such a system acts as an interface between the ultimate user's view of the data (logical view) and the hardware of the system. The DBMS is, therefore, a mapping between the logical structure of a data base and its physical reality (Figure 1).

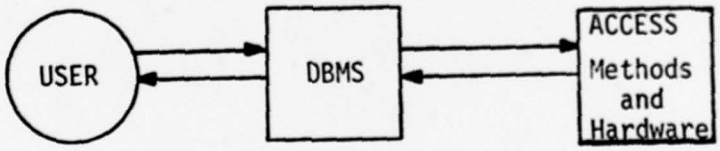


FIGURE 1

PICTORAL REPRESENTATION OF THE DBMS INTERFACE

3. Logical Data Independence

A data model is logically independent if the programs which access the particular data base are unaffected by logical changes in the data model.

4. Data Integrity

A data model has data integrity when at all times there exists no inconsistency in the data base. A data model looses integrity when any data items which should represent the same fact do not contain the same information, i.e., when they are inconsistent. It is a problem of data base design to insure that inconsistency is eliminated, or is always kept at a minimum level.

5. Hierarchical Data Model

a. The hierarchical data model has the following characteristics: [1].

1) There is a set of record types {R1,R2,...,RN}.

2) There is a set of relationships connecting all record types in one data structure diagram.

3) There is no more than one relationship between any two record types, R_i and R_j.

4) The relationships in the data structure form a tree with all arcs pointing towards the leaves.

5) Each relationship is 1:N in the direction of the leaves of the tree, and is total -- that is, for every R_j record occurrence there is exactly one R_i record occurrence connected to it, if R_i is the Parent of R_j in the definition tree.

b. R_i is said to be an ancestor of R_j iff $i < j$ and R_i lies on a path to R_j. R_j is a descendant of R_i if the same conditions hold.

c. Operations on a hierarchical data model are:

GET DESCENDANT (Record)

GET ANCESTOR (Record)

GET NEXT (Record)

In some systems, requests may be made non-procedurally using statements of the form: ACTION A WHERE P. A is a retrieval of some subset of record types contained in the model, and P is a hierarchical predicate which qualifies elements and records along a common (family) path. A multi-branch (path) qualification is generally illegal because cases arise where processing of qualifications cannot continue unambiguiously [11].

Figure 2 shows an example of the occurrence structure of a hierarchical data base with 4 record types. Type A is a root and is located at level 0.

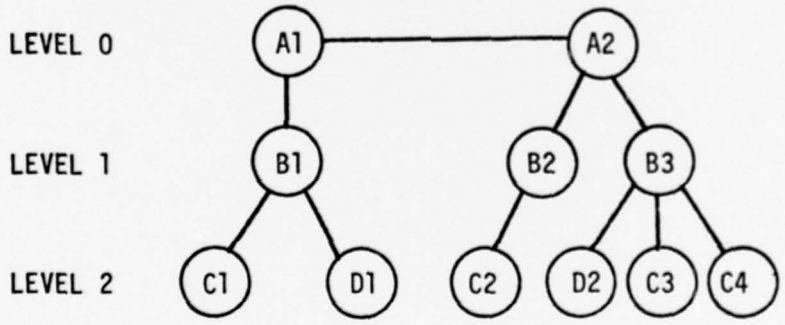


FIGURE 2

HIERARCHICAL ORGANIZATION

In order to retrieve the subtree A1, and then retrieve C1, the following sequence of commands could be followed in a low-level navigation language:

GET NEXT A.

GET DESCENDANT B.

GET DESCENDANT C.

Notice that while a record of type A may be retrieved using a GET A WHERE <B1 has condition> AND <C1 has condition>, a request of the form GET A WHERE <C1 has condition> AND <D1 has condition> is illegal because it spans two hierarchical paths. This limitation has direct affect on the final structure of the relational data base (CHAPTER III).

The term repeating group (RG) is used to describe structures for storing multiple sets of data values, and to link the levels of the hierarchy. In Figure 2A, B, C, and D are examples of repeating groups.

The Presidential Data Base [1] is presented in Figures 3 and 4 in hierarchical form. Figure 3 provides the definition tree or schema, Figure 4 is an example of how some sample data would appear for two presidents, Nixon and Carter.

6. Relational Data Model

The Relational Data Model is based on the mathematical theory of relations. It was first presented by Codd in 1970 and has since been a topic of great discussion. The following terms are appropriate to relational data bases [9].

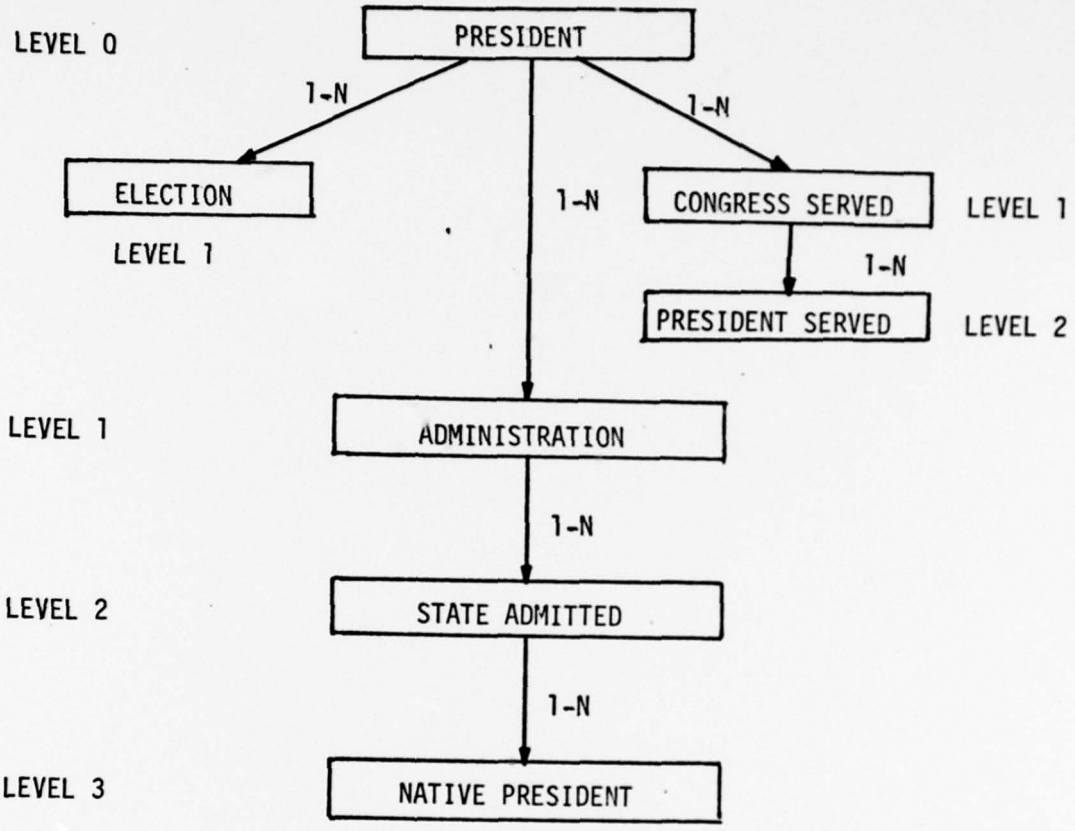
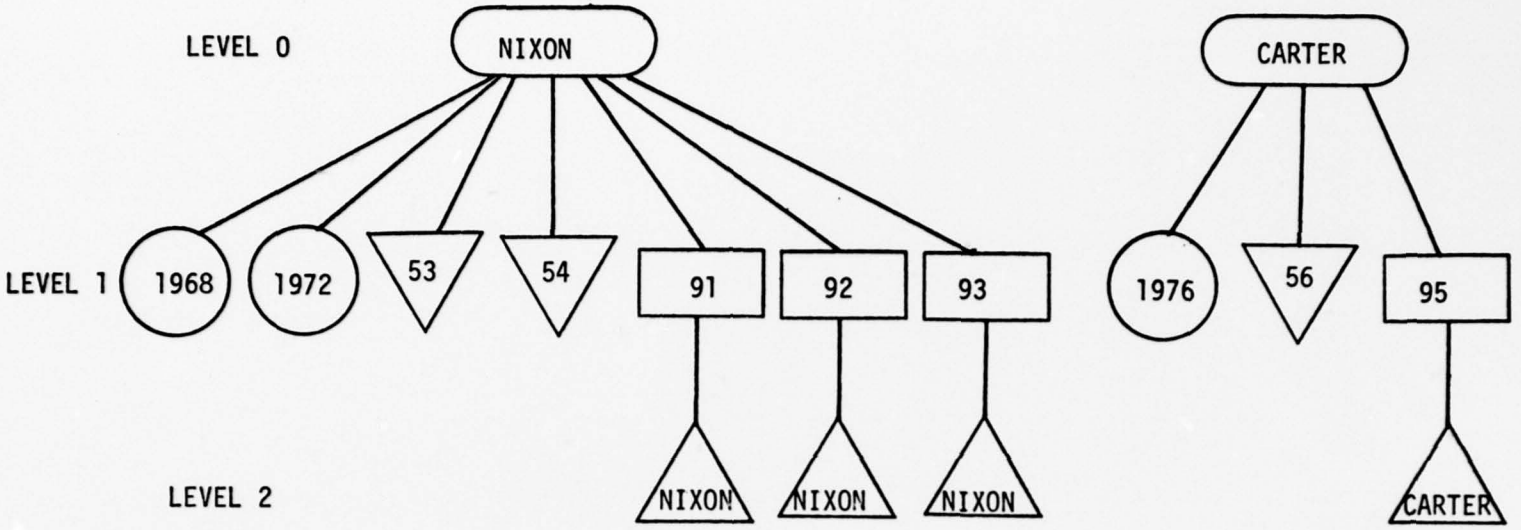

LEVEL 0
PRESIDENT
1-N
1-N
1-N
ELECTION
LEVEL 1
CONGRESS SERVED
LEVEL 1
1-N
PRESIDENT SERVED
LEVEL 2
LEVEL 1
ADMINISTRATION
1-N
LEVEL 2
STATE ADMITTED
1-N
LEVEL 3
NATIVE PRESIDENT

FIGURE 3



NO LEVEL 3

RECORD TYPES

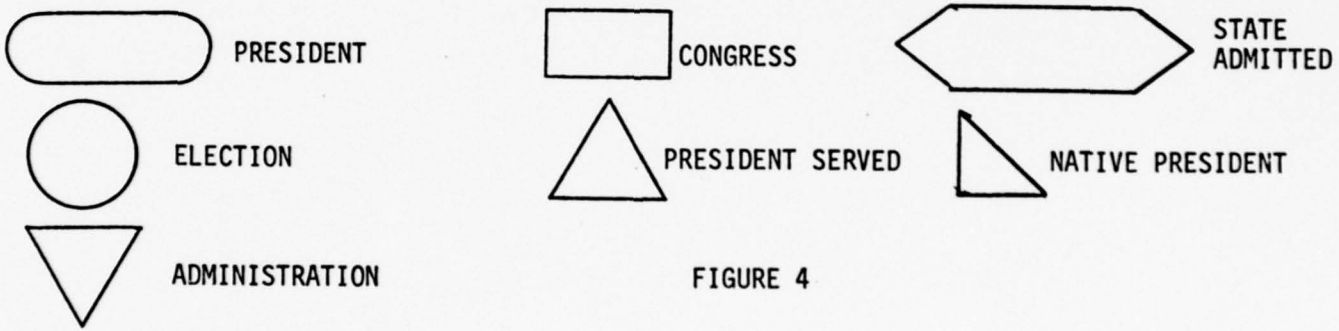


FIGURE 4

Given sets $D_1, D_2, \ldots, D_N$ (not necessarily distinct), a relation, or table, R, is a set of n-tuples each of which has its first element from D_1, its second element from D_2, etc. The sets D_i are called domains, or columns. The n-tuple is alternately referred to as row. The number n is called the degree of R and the number of tuples, or rows is called its cardinality. A relation is alternately referred to as a table.

The following properties derive from the definition of relation:

1) No two rows (tuples) are identical.

2) The ordering of rows is not significant.

3) The ordering of columns is not significant as long as the column headings are presented with the respective rows.

Each relation has a name, and each column is named. The columns of a table are called attributes. Elements of a tuple are called components or elements. Elements with similar attributes are elements with similar data types (integer, real, alpha-numeric, etc.). All elements of a column must be of similar data type. An example of a relation is shown in Figure 5.

< table name >

<column name>$_1$, <column name>$_2$ ... <column name>$_N$

<component$_{11}$, component$_{12}$, ... , component$_{1N}$>

<component$_{21}$, ... , component$_{2N}$>

⋮

<component$_{M1}$, component$_{M2}$, ... , component$_{MN}$>

FIGURE 5

TABULAR REPRESENTATION OF A RELATION WITH DEGREE N AND CARDINALITY M

The presidential data base is presented in relational form in Figure 6. The structural representation is more compact than the hierarchical model, the "structure" of the whole system being contained in the tables themselves. Such organization provides for both physical and logical data independence, reducing the complexity of the system by hiding it from the user. It is beneficial to the casual user because he views all relations and all components on equal levels. Essentially, to the user, the data base is a flat file from which any component can be selected at any time.

A relation is said to be normalized and to be of the first normal form if each element in each row is a non-decomposable data item (ergo, a number or character string, not a table name). A column B of a table R is said to be functionally dependent on a column A of R, or a set of columns A of R, if at every instance of time each B entry is associated with only one A entry. A is said to determine B, and A is called the determinant of B.

A column or set of columns whose values uniquely identify a row is called a key. Tables may have more than one key per row. In such cases one of the keys is designated as the primary key.

A relation R is said to be in the second normal form if and only if all of the non-key domains (columns) of R are functionally dependent on the primary key of R. A normalized relation R is said to be in the third normal form if and only if it is in second normal form and all of the non-key domains of R are mutually independent. That is, no column which is not a member of the primary key is

ELECTIONS WON

YEAR	WINNER-NAME	WINNER-VOTES
1952	Eisenhower	442
1956	Eisenhower	447
1960	Kennedy	3303
1964	Johnson	483
1968	Nixon	301
1972	Nixon	520

PRESIDENTS

NAME	PARTY	HOME-STATE
Eisenhower	Republican	Texas
Kennedy	Democrat	Massachusetts
Johnson	Democrat	Texas
Nixon	Republican	California

ELECTIONS LOST

YEAR	LOSER-NAME	LOSER-VOTES
1952	Stevenson	89
1956	Stevenson	73
1960	Nixon	219
1964	Goldwater	52
1968	Humphrey	191
1968	Wallace	46
1972	McGovern	17

LOSERS

NAME	PARTY
Stevenson	Democrat
Nixon	Republican
Goldwater	Republican
Humphrey	Democrat
Wallace	Am. Indep.
McGovern	Democrat

WIFE

CANDIDATE NAME	WIFE NAME
Nixon	Pat
Stevenson	Ruth
Goldwater	Ruth
Humphrey	Agnes
Wallace	Melody
McGovern	Jane

FIGURE 6

PRESIDENTIAL DATA BASE IN A RELATIONAL FORMAT [1]

functionally dependent on any column or group of columns other than the primary key.

Operations in a relational DBMS are defined in terms of either a nonprocedural language or an algebra. The relational algebra normally consists of the following operations:

a. PROJECTION

PROJECT <table name>.<column list>

The projection operator returns the specified columns of the given relation (table name) and eliminates any duplicate rows from the result.

b. SELECT

SELECT <table name>[column name<relational-operator><value>]

where <relational operator> is one of the following operators: > < $\geq$ $\leq$ = $\neq$. The selection operator selects only those rows of a given relation which satisfy the specified condition. The <value> may be either a constant or a row element.

c. JOIN

JOIN <table name>.<column name> TO <table name>.<column name>

JOIN A.<col> TO B.<col>

The JOIN operator returns a relation formed by concatenating a row of the first table (A) to a row of the second relation (B) wherever values in the specified columns are equal. If any row in table A matches more than one row in B, it is concatenated with each of them,

forming as many new rows as the row in table A matched in table B.

d. SET OPERATIONS

UNION <table name><table name>

INTERSECT <table name><table name>

DIFFERENCE <table name><table name>

Each of the set operations returns the appropriate set theoretic result in the form of one relation. The operand relations must have compatable sets of attributes, i.e., the two relations must have the same degree and columns of a similar type.

e. DIVISION

<table name>.<column name>$_1$,<column name>$_2$(A)

DIV <table name>.<column name>(B)

Division returns a quotient relation which consists of <column name>$_1$ entries of A. A <column name>$_1$ element, say X, is in the quotient if and only if for every <column name>$_i$ element of B, there exists a row of A with <column name>$_1$ equal to X and <column name>$_2$ equal to the <column name>$_i$ of B. [1]

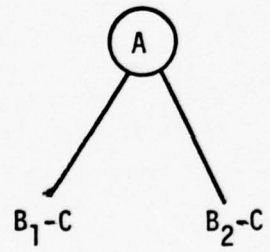
The non-procedural language takes the form of a relational calculus or of a "mapping oriented language". The calculus was proposed by Codd [8]. Mapping oriented languages include SEQUEL [3,6] and SQUARE [4]. Additional information on relational systems can be found in [9] and [7].

7. Relational System vs. Hierarchical System

This section is a limited discussion of aspects of comparison and difference between the relational and hierarchical approaches.

As discussed earlier, to the user the relational model has only one level. Since all items exist at this level, each table in the data base may be used to qualify any query about another table. In the hierarchical model the universe of operations is more limited. Clemons [7] calls this a lack of 'directional bias'. One reason for using the hierarchical model could be an organization which provides for mutually exclusive cases of record occurrences, or mutually exclusive access paths. Reasons for using a relational model include data bases where the universe of operations varies with time.

Any relational system implemented in terms of a hierarchy must be constructed in terms of a hierarchy. The expression of a hierarchical data base in terms of a relational system is possible (see Figure 7) simply by creating a new column for each level above a particular level and repeating the data values as many times as there are particular elements.



A	B_1	C
A	B_2	C

FIGURE 7

EXPRESSION OF A HIERARCHY IN TERMS OF A RELATION. C REPRESENTS COMPONENTS OF REPEATING GROUP B

The expression of the relation in terms of a hierarchy is not as straight-forward as expressing a hierarchy in terms of relation, and is one of the problems addressed by this thesis in CHAPTER III.

Data integrity is another issue in which there are differences between the relational model and the hierarchical model. The relational model is subject to loss of data integrity if relations are not normalized. Hierarchical data bases are subject to loss of integrity but the nature of the hierarchy allows designs which provide an integrity. A conflict arises in some possible expressions of a normalized relational data base in hierarchical form, particularly that form proposed by Smith and Smith [19] unless the precautions proposed in their paper "Data Base Abstraction" are followed.

8. Conclusion

This chapter has defined terms that will be used extensively in later chapters. It discussed the hierarchical and relational models of a data base and mentioned two points of concern to be discussed in CHAPTER III.

The remaining chapters discuss the problem of defining a relational data base in terms of a hierarchy and the implementation algorithms to be used in the construction.

CHAPTER III

PROBLEM DEFINITION

1. Problem

The problem of this thesis is to study the design of a relational data base management system as a front end to an existing hierarchical data base management system.

2. Constraints

a. The underlying hierarchical system is to be transparent to the user.

b. A Relational Algebra and a subset of the query language SEQUEL [6] are to be implemented. The SEQUEL subset will be implemented in terms of the relational algebra.

c. This system will be designed for relatively small data bases with a limited application - student data bases. It is a demonstration of capability rather than a massive data base project.

d. Because of the student environment, a user should be able to design a data base, load it and run it with no outside help from a data base administrator.

e. As much as possible the front end system should make use of the operations available in the hierarchical model. The "front end" is used only as a translator and to implement all relational operations in terms of operations

and queries on the underlying structure. Only when it is impractical to use the available functions, will any new functions be created.

f. The front end should support any relational data base organization.

g. The front end system designed will not be a relational interpretation of any hierarchical organization. It will rather use the hierarchical data base management system as a low level mechanism to manipulate relational data structures which are defined in terms of the hierarchy.

3. Goals of the Project

a. Exploring (in terms of implementation) the mapping relationships between a relational data base and a hierarchical data base.

b. Identifying both rejected data structures and inefficient methods so that any future implementations will have a knowledge base from which to start.

c. Pointing out areas for further research.

4. The Underlying Hierarchical System

MRI System Corporation's SYSTEM 2000 Version 2.4 for the Control Data 6000/CYBER hardware environment is the underlying DBMS. SYSTEM 2000 (S2K) is a powerful general purpose data base management system which provides the user with the ability to design a data base

to fit his particular requirements. SYSTEM 2000 provides both a natural language query capability and a procedural language interface with the basic system. The procedural language can be PL/1, FORTRAN or COBOL. In the front end implementation to be discussed, FORTRAN was chosen as the implementation language because of its word manipulation capability. The procedural language interface (PLI) was chosen over the natural language interface because using a natural language query requires that the query, the generator, and S2K be swapped as they alternate execution, and because it turns out that the natural language available is no more powerful in terms of the relational operations than the FORTRAN procedural language interface.

Several relevant aspects of natural language and PLI are now discussed [17]. The section finishes with additional constraints imposed by PLI.

The Define module of SYSTEM 2000 organizes the elements of the data base into repeating groups. Levels of information are specified within the define sequence by declaring a "component" (i.e., a data item) to be in another component.

Procedural Language is designed for systematic work with a data base rather than specific one time queries. The interface consists of the programming language FORTRAN and additional subroutines, which are a subset of the routines supplied by MRI Systems Corp.

An important concept is that of schema. A PLI schema is a group of variables which represents particular parts of, or all elements of, a repeating group. The schema serves as an input/output buffer

for the PLI program. The S2K defined component name is used as a variable in the schema.

A logical entry is made up of one level zero data set (RG) and any descendant data sets, where data set is defined as a unique set of data logically defined by the data base definition as belonging to the same repeating group. A logical entry is therefore all of the information about one of the major records stored in the data base, or a complete data tree.

A family tree is defined as all of the descendant data sets, (RGs) of a logical entry. In Figure 2 there are two families represented, one with root at A1 and another with root at A2. Each of the repeating groups B, C, and D are components of the repeating group A. Each occurrence of A and its descendants composes a logical entry.

SYSTEM 2000 PLI provides the following operations which will be used extensively in the following chapters.

GET1 <schema> retrieves a unique occurrence of a data set and places its values into the specified schema. GET1 may be used with a WHERE clause which qualifies the unique schema to be retrieved, or without the WHERE clause, in which case the first occurrence of the specified schema that the DBMS encounters will be returned.

A WHERE clause takes the general form of a sequence of qualifications of the members of a family tree. Each qualification has the form <name><op><value>, where <name> is the S2K component name referenced in a schema, <op> is one of the relational operators

equals, less than, less than or equal, greater than, greater than or equal, exists or fails to exist, and <value> is an actual "quantity" input to the S2K routines by the user program.

LOCATE <schema> WHERE <where clause> is a routine which locates, but does not retrieve any data. It locates all occurrences of the named schema where the <where clause> conditions hold, and constructs a list of pointers to those occurrences. In order to retrieve data sets which have been located by a LOCATE statement, the GET function must be used.

GET <schema><occurrence parameter> can retrieve any of the data sets located by a previous LOCATE statement. The <occurrence parameter> includes FIRST for the first schema in the list, NEXT for the next schema in the list, LAST for the last of the previously located schemas, or S2KCOUNT for retrieving a data set which is a specified number (greater than zero) of data sets removed from the last retrieved schema.

Position in a family is established by GET1 and/or GET commands, while LOCATE is used to collect all families of a qualified class. Other subroutines are GETA for get ancestor and GETD for get descendant. Each of the subroutines mentioned, with their power and limitations, have a direct bearing on the structure of the data base. Further information about all of these operations, as well as PLI in general, is contained in the SYSTEM 2000 Reference Manual [17].

Natural language provides more powerful features than procedural language in that a single statement can both qualify and

output many types of data. The important difference which bears on this implementation problem is that of multi-family query power. Natural language provides a "HAS" operator which enables the user to ask many branched questions. The HAS operator basically allows the following qualifications:

ACTION X WHERE RG EQ <value> AND RG HAS
<some component> = <some value (constant)>.

The HAS operator does not allow qualifications of the type <component> = <component>, in the Control Data Implementation. These types of qualifications are exactly what is needed in order to implement relational algebra operations i.e., JOIN would be expressed as a retrieval of subsets of two trees, where specified components are equal. The structure of the query does not permit such qualifications. a JOIN is forced to the repetitive activity of retrieving one record of a table and joining it to the second table of the join. Such repetition is best suited to a procedural language.

It is possible to achieve some relational operations using the user-defined function capability of SYSTEM 2000. This capability does not extend to processing sequences of commands, for which a driver is needed. Additionally, the current implementation at UT Austin (Version 2.4) requires that if natural language queries are generated there must be a generator program and provision to roll the DBMS and the generator in and out of execution, and to generate successive requests and store intermediate results in the SYSTEM 2000 data base.

Such methods do not grant any particular power over procedural language. They also require more time because the roll in/roll out start up time must be added to execution time. Procedural language is elected over natural language because it should execute in less time and because in the context of this project, no additional power is provided by natural language.

It is desired to make as much use of the available PLI functions as possible. At the same time it is necessary to not overload the system with excessive PLI calls. For this reason, the size of a table (relation) is limited to the number of elements which can be qualified in any one subroutine call.

PLI permits 10 schemas to be referenced in a call and 25 conditions to be set. Because the organization selected provides for each domain value (column) of a relation to be in a separate schema and referenced in that schema, and because each qualification requires the table name to be referenced, the degree of a table is limited to 9. This limitation does not affect the flavor of the relational system, it does not simplify the mapping function used between tables and hierarchical elements, and it does not simplify the relational algebra algorithms.

For ease of manipulation in FORTRAN, the number of rows in a table (relation) is arbitrarily limited to fifty. Since the data bases used should be small, and since using more space per table increases core size in an already large program, this limitation is made.

5. Conclusion

This chapter discussed the problem, the goals of the project and the limitations placed on the system from the outset. The next chapter concerns the actual choice for data base design. The procedural language will be used for the program, the degree of each table will be limited to 9 and each table's cardinality will be limited to fifty.

CHAPTER IV

THE UNDERLYING HIERARCHY

An underlying data base structure design is desired which, within the criteria of the previous chapter lends itself to the efficient use of PLI, is as simple as possible, has no excessive redundancy or wasted space, and will support any normalized relational data base.

The first design considered is to have table organization only at the element level. All information about where an element is located is carried with it, and each element is immediately available at level zero of the data base. A definition (in S2K terms) of the organization is:

C1*ELEMENT(RG)

C2*VALUE(INTEGER)

C3*TABLE NAME(INTEGER)

C4*ROW NAME(INTEGER)

C5*ROW NUMBER(INTEGER)

All relational operations can be implemented in terms of this schema; however, the schema is redundant and requires that the driver do excessive work. In order to examine a particular row, for example, the row and table must either be qualified (using LOCATE) with a series of GET operations done to retrieve the elements, or a sequence of GET1 operations must be performed.

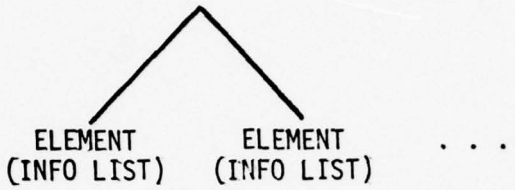
An algorithm for a JOIN of two tables, for example, is straight-forward.

JOIN A.COL_x TO B.COL_y

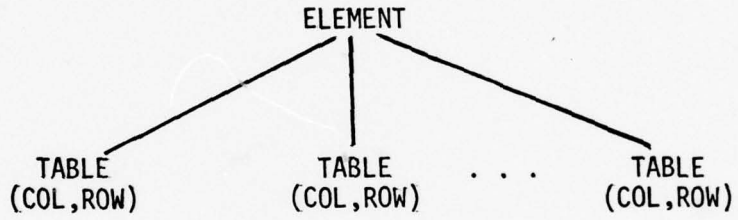
1. NUMROWS = 1 ROW COUNT = 1
2. LOCATE all elements of table A, Row N and place them in "HOLDING BUFFER" (ROW COUNT). If any data qualified GOTO 3 ELSE done with JOIN
3. LOCATE all of table B.COL_y elements which are equal to table A, Row N, COL_x. IF any data qualified GOTO 4 ELSE GOTO 6
4. COUNT = ROWCOUNT. GET all elements of the NEXT ROW of table B which qualified, and place them in HOLDING BUFFER (COUNT). IF end of data GOTO 5 ELSE COUNT = COUNT + 1 GOTO 3
5. Repeat all elements of ROW (NUMROW), TABLE A (COUNT-ROWCOUNT) times in the HOLDING BUFFER.
   ROWCOUNT = COUNT + 1 GOTO 6
6. NUMROW = NUMROW + 1
   GOTO 2

Since the relation and the hierarchy are equivalenced on the element level the driver is forced to operate at the element level. An entire row or table can be located and any element or sequence of elements can be accessed. This organization does not, however, make use of the hierarchical structure, and is redundant. Obviously some redundancy can be tolerated, but carrying three extra values per element, added to two pointers per repeating group [18] is excessive.

The more information carried per record, the more work can be done with that record. The structure of the hierarchy should be utilized and table and row association assigned by position. Note that ruling our structures of the form:



also rules out having other element level 0 organization. For example a structure with only one of each



element value and using repeating groups for all relations of which it is a member is ruled out because while it is a less redundant structure than the first one considered, the structure still must contain redundant column and row information.

The second schema considered, is organization by table name and column name. The S2K definition is, for example:

1*DATA BASE INFORMATION

2*TABLE AND TABLE INFORMATION (RG IN 1)

3*COLUMN AND COLUMN INFORMATION (RG IN 2)

4*ELEMENT (RG IN 3)

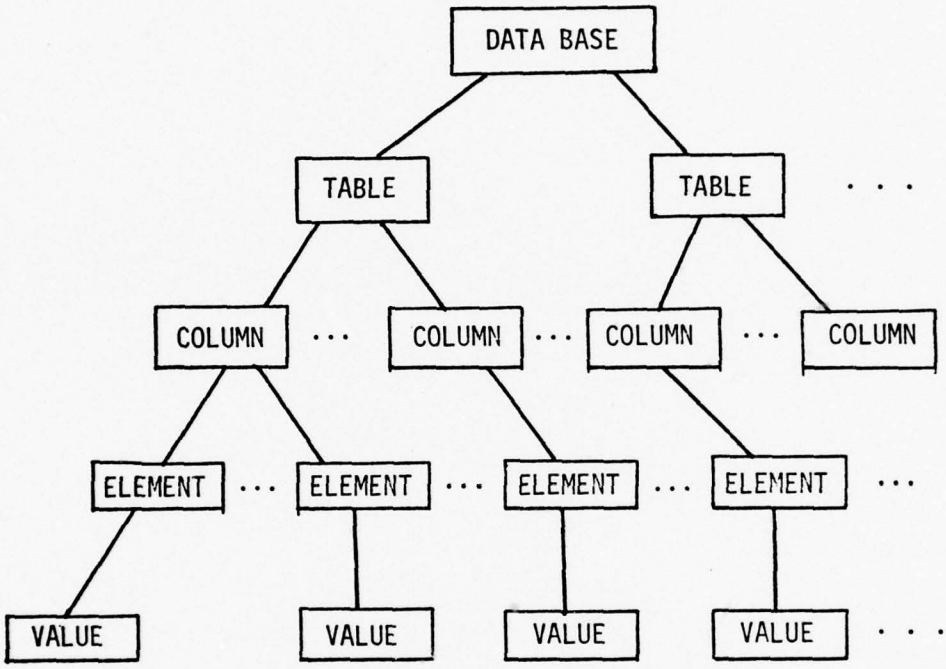
5*REAL VALUE (RG IN 4)

6*INTEGER VALUE (RG IN 4)

7*ALPHA VALUE (RG IN 4)

8*ROW NUMBER (INTEGER IN 4)

An illustration of the organization is:



While this schema allows access to any column, for relational operations, the organization still requires that a series of GET operations be performed in order to retrieve a row. The PLI subroutines should be utilized to a greater extent than this schema allows, because only one field of the 10 available in a WHERE clause is used. Locating can only be done one element at a time and operations such as intersection, where all elements in a row should be qualified at once against

all elements in rows of a target relation, are many branch operations and are illegal.

Organizing by row, using a series of equal level repeating groups such as Figure 8, also leads to multibranch queries,

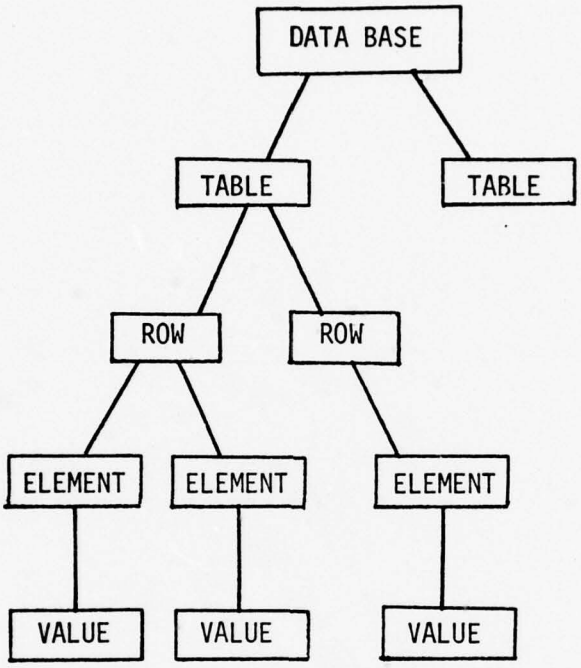


FIGURE 8

since organization on the element level is the same as that of schema 2. A request to locate a row by more than one element is multibranch and is illegal.

At this point the HAS operator of natural language which allows multibranch queries of the type desired in schemas 2 and 3 must be considered again. In order to qualify a row, however, there must be values by which to qualify. The elements must come from one table

and the qualification must occur one row at a time. That is, intersect table A with table B has the form:

LOCATE ROWS WHERE DATA BASE HAS ((TABLE NAME EQ A) HAS

((ROWS EXISTS) AND (ROWS HAS ELEMENTS EQ (DATA BASE HAS ...))))).

The problem with the above expression is that since one and only one value is expected after an EQ, the expression is illegal. Comparison of the elements in one table with elements in a second table must be accomplished by repeatedly removing each element in one of the tables and using the element to qualify the second table.

The fourth schema is an attempt to avoid the problems of iteration encountered in the first three schemas. Following the general philosophy presented by Smith and Smith [19], a very general hierarchy is designed (Figure 9).

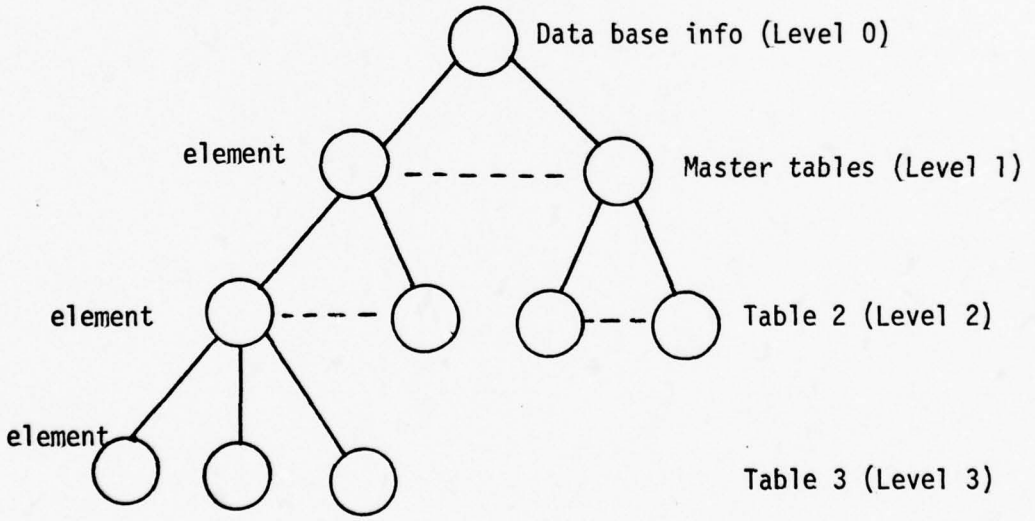


FIGURE 9

This schema involves having each element of each table be a potential key through which to access all tables "below" it. Relations are input along with key values and/or some indication as to what columns will have the same attribute, but different names. Load time processes involve positioning in the data base according to the key elements of a row i.e., (WASHINGTON, MARTHA ...) would involve positioning at the Washington element of the level above the wife level and inserting the wife row at the next level. The number of elements in such a hierarchy becomes complicated quickly. By the sixth level of relations, there are no more SYSTEM 2000 component numbers available. This model would construct too large a definition for the hierarchy.

A problem thus far has been that elements of a table must be retrieved by the PLI program in order to perform an operation with another relation in the data base. The relational algebra deals with whole classes of elements and relates them to other classes of elements. The hierarchical system's languages assume that the user is interested in qualifying one class or set of classes by some known value or values. The structure is assumed to model the set of interesting queries. The system works its way down a tree or group of trees with only one comparison per record element in mind. It is not structured to handle class to class comparisons.

The result of this observation is to return to a simpler schema, which yields a smaller data base definition and a smaller PLI program.

The next schema considered is an expansion of an earlier schema. It involves assigning a separate component value to each of the 9 possible elements of a row. A schema which utilizes repeating groups at the same level in the row repeating group (Figure 10) is illegal because comparing more than one component at once is a multi-family qualification.

A modification of this schema can eliminate the multifamily problem (Figure 11). This schema provides that each row contain a single family of elements. The first element is contained as a repeating group in the row, the second element as a repeating group in the first, the third as an element in the second, and so on to the ninth element. Now an entire row can be qualified by a PLF query with only one locate.

However, now the size consideration enters as the number of elements per repeating group are considered. Each repeating group requires 2 pointer words (in the CDC 6600/6400) in addition to the words containing data values. Using only one data word record (for a seven character algebra value and a 15 digit decimal integer) by the fourth repeating group, 12 words have been used for pointers and values. By the last or ninth repeating group, a total of 27 words have been used by a schema with all elements as components of one repeating group (ROW).

The break even point for word size occurs at record number four where twelve words were assigned versus a constant total of eleven in a one repeating group row system. This organization allows

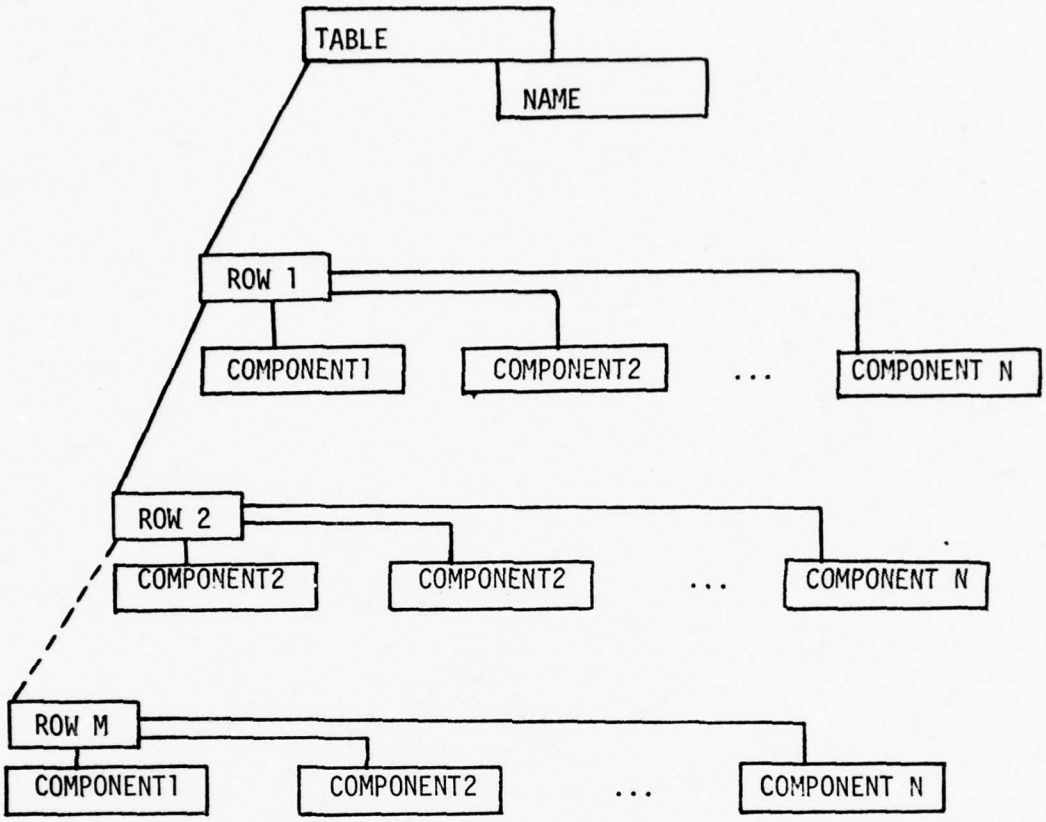


TABLE
NAME
ROW 1
COMPONENT1
COMPONENT2
...
COMPONENT N
ROW 2
COMPONENT2
COMPONENT2
...
COMPONENT N
ROW M
COMPONENT1
COMPONENT2
...
COMPONENT N

FIGURE 10

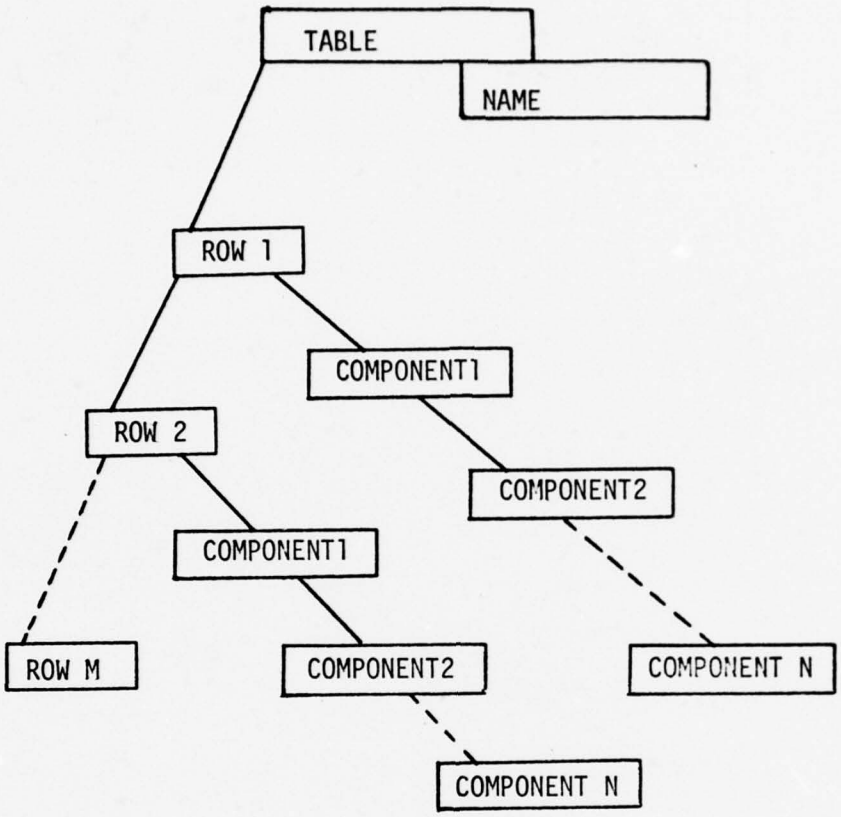


TABLE
NAME
ROW 1
COMPONENT1
ROW 2
COMPONENT2
COMPONENT1
ROW M
COMPONENT2
COMPONENT N
COMPONENT N

FIGURE 11

retrieval of an entire row (the first schema allows only a sequence of GETD operations to retrieve a row). It will make the PLF program smaller, and will simplify it. Because this restriction does not cause further restrictions to be placed on the system, the schema shown in Figure 11 was chosen for the data base for the project.

There will be no difference between the two schemas in Figures 10 and 11 for any LOCATE statement. The difference in operation arises in a "GETROW" (CHAPTER V) call. In the first schema (repeating groups in repeating groups), "GETROW" would make the successive GETD calls and return a value to a specified place. In the program schema GETROW is one call to GET, which places all elements in a PLI program buffer. The presidential data base is displayed in Figure 12 using the schema of Figure 11.

This chapter has discussed possible schemas for use in the underlying hierarchy. It was discovered that in order to accomplish relational operations there must be a sequence of retrievals and qualifications on the retrieved elements. This knowledge leads to schemas which provide for both small definition space and ease of access for both row retrieval and qualification.

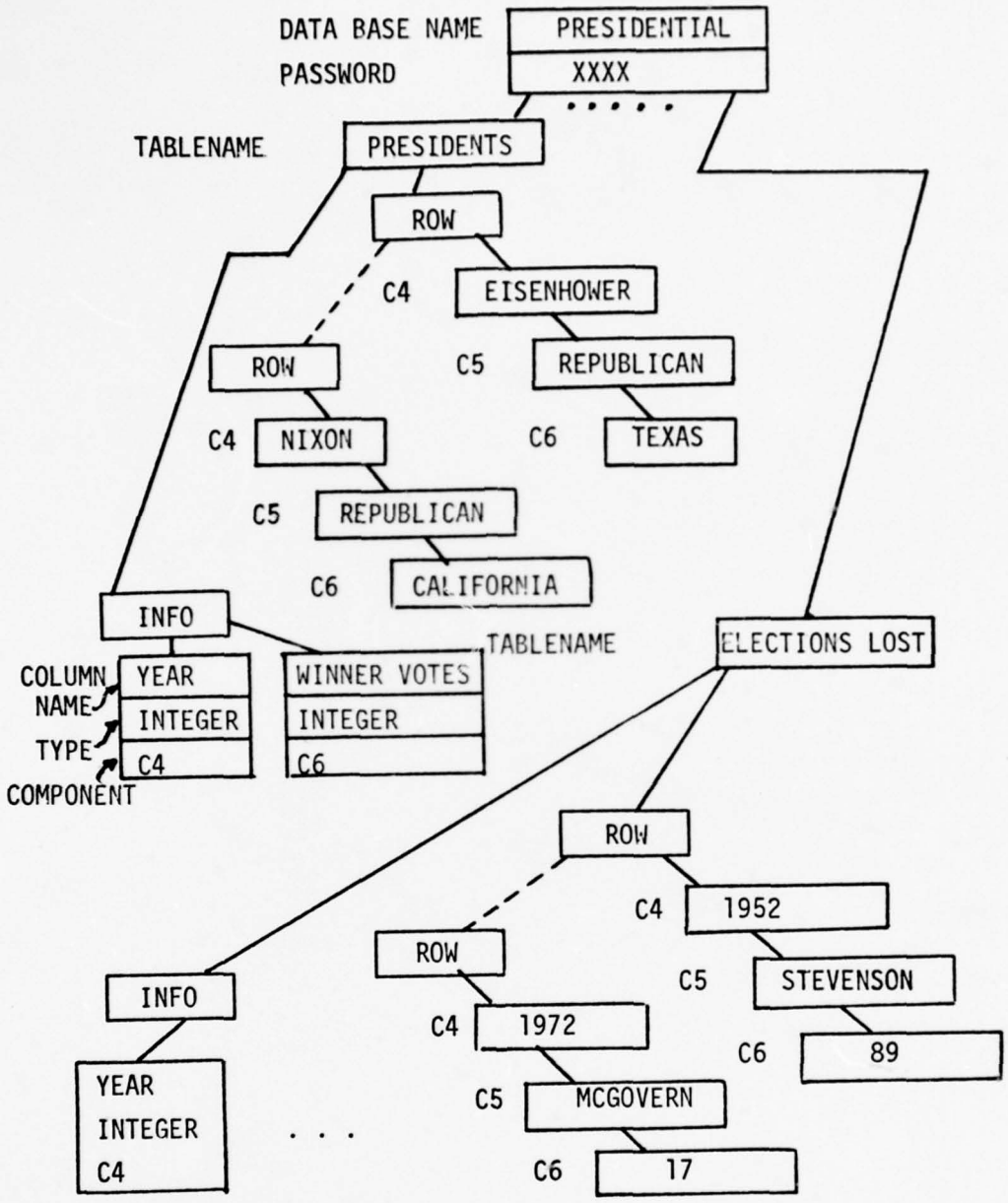


FIGURE 12

PRESIDENTIAL DATA BASE IN THE SELECTED SCHEMA

CHAPTER V

ALGORITHMS FOR THE INTERFACE

This chapter discusses the interface program. It presents algorithms for the relational algebra operations join, division, restriction, union, selection, projection, and difference. It discusses the implementation of the SEQUEL subset in terms of sequential applications of the algebra operations.

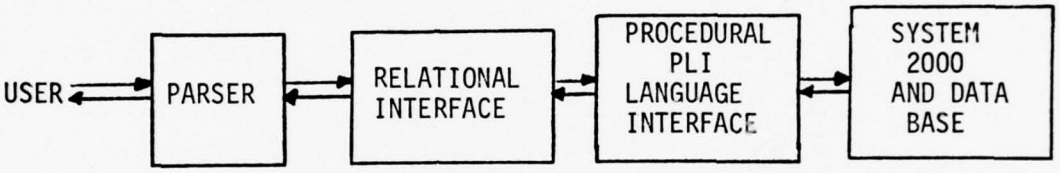


FIGURE 13

This is a four level system (Figure 13). The outer level is a modification of a Pascal parser which was written for use with the University of Texas implementation of the BOBSW parser generator program [5]. The parser drives the relational interface, which drives the PLI interface programs, which perform the data base manipulation by driving SYSTEM 2000. The system is designed so that it can be overlayed. The parser and the main overlay reside in core continually.

A hierarchical definition tree of the data base is shown in Figure 14. To aid in understanding the processing of the algorithms, the table sub-tree is expanded in Figure 15.

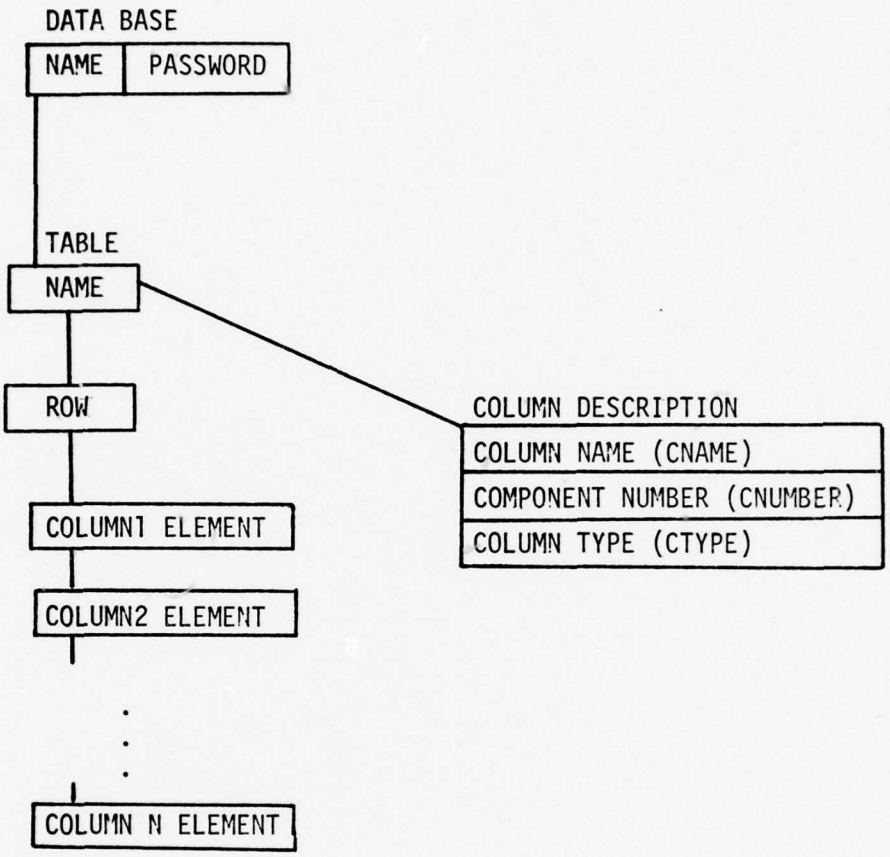

DATA BASE
NAME
PASSWORD
TABLE
NAME
ROW
COLUMN1 ELEMENT
COLUMN2 ELEMENT
COLUMN N ELEMENT
COLUMN DESCRIPTION
COLUMN NAME (CNAME)
COMPONENT NUMBER (CNUMBER)
COLUMN TYPE (CTYPE)

FIGURE 14

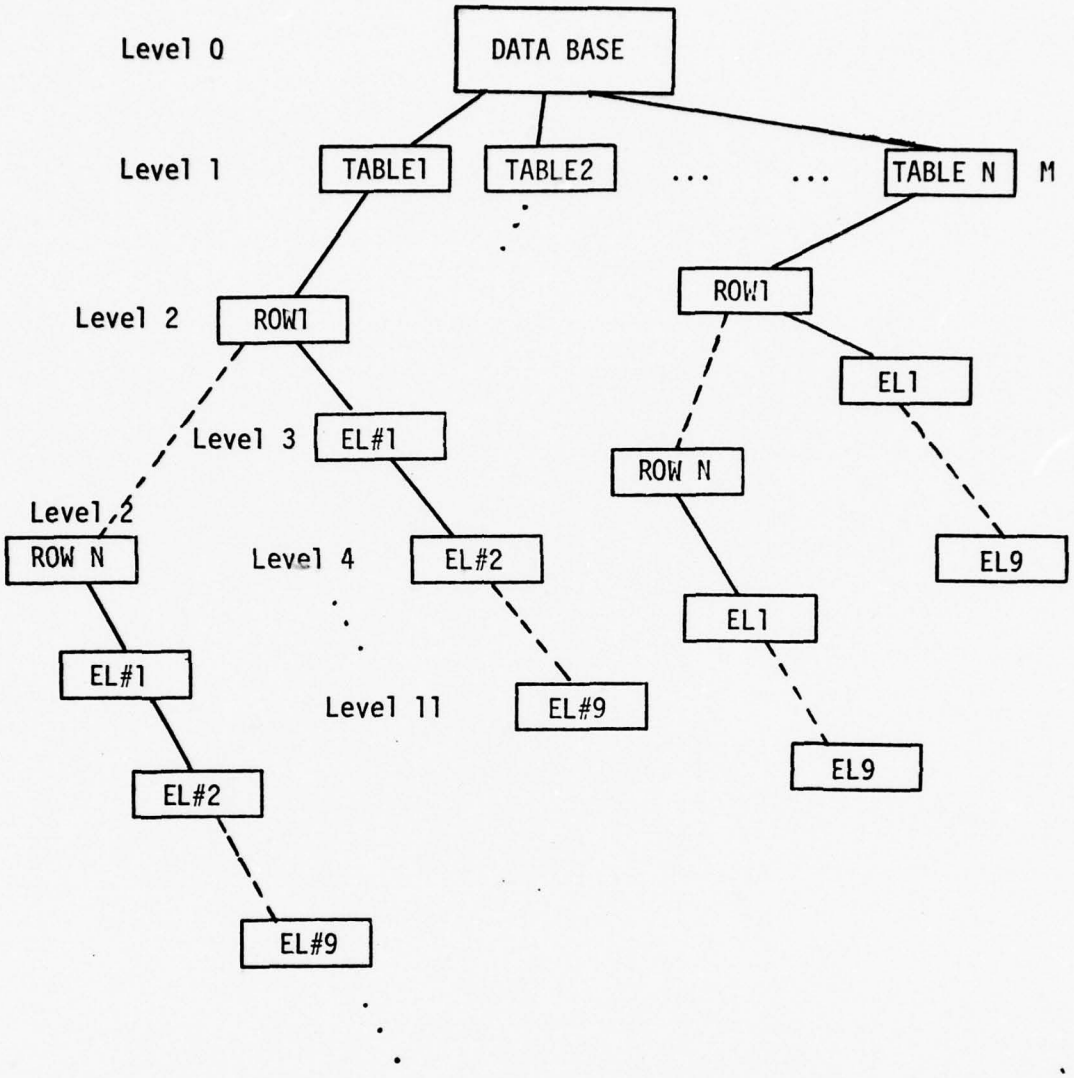

Level 0
DATA BASE
Level 1
TABLE1
TABLE2
...
...
TABLE N
M
Level 2
ROW1
ROW1
EL1
Level 3
EL#1
ROW N
Level 2
ROW N
Level 4
EL#2
EL9
EL1
EL#1
Level 11
EL#9
EL9
EL#2
EL#9

FIGURE 15

The relational interface duplicates the table name and column description information for rapid use by the semantic routines. The interface table information is kept in a linked list of records, one of which is shown in Figure 16. Name, number and type fields for a specific column hold its print name, SYSTEM 2000 component number, and type, respectively.

The semantic routines maintain two push down stacks, one for tables and one for relational operators. Figure 17 shows the contents of an entry on each stack, as well as its record structure. Each semantic routine expects to find the correct number of arguments on the table stack. If the correct number of arguments is not found, an error is returned and the statement terminated.

Unless a statement calls for a specific projection (a PROJ in the algebra or a SELECT <col list> in SEQUEL), all columns of a table in the stack are kept. This allows ANDS and ORS in SEQUEL statements to be processed as table intersection and union. The use of intersection and union in this context is demonstrated in an extension of an example from Boyce and Chamberlain [6].

EMP

NAME	DEPT	MGR
John	Shoe	Bob
Fred	Shoe	Frank
Fred	Toy	Bob
Bill	Toy	Frank

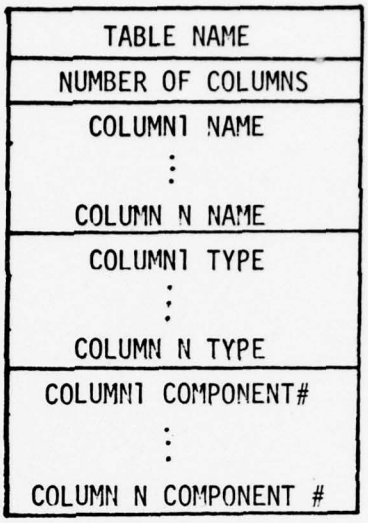


FIGURE 16

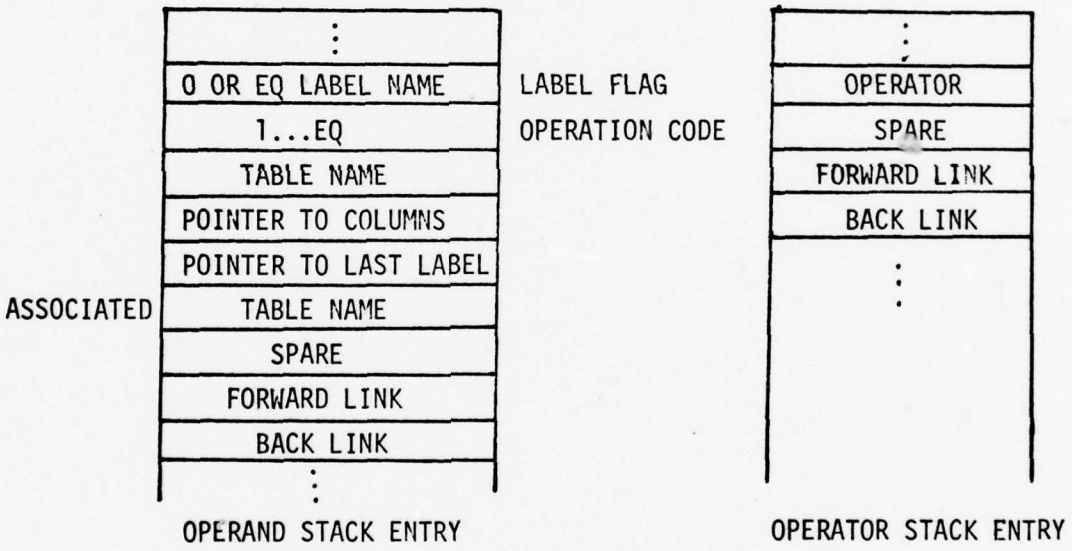


FIGURE 17

QUERY 1 SELECT EMP.NAME WHERE DEPT="shoe" AND MGR= "Bob"

QUERY 2 SELECT EMP.NAME WHERE DEPT= "shoe" ∩

SELECT EMP.NAME WHERE MGR= "Bob"

Query 1 processes as:

Selection of all "shoe"=DEPT rows:

NAME	DEPT	MGR
John	shoe	Bob
Fred	shoe	Frank

Selection of all "Bob"=MGR rows:

NAME	DEPT	MGR
John	shoe	Bob
Fred	toy	Bob

Intersection of the two tables yields the correct answer:

NAME	DEPT	MGR
John	shoe	Bob

Query 2 processes as:

Selection of all "shoe":DEPT rows and projection of NAME:

NAME
John
Fred

Selection of all "Bob":MGR rows and projection of NAME:

NAME
John
Fred

This intersection yields the correct values of John and Fred.

The design of any algorithm for use with a procedural language has several limitations which must be considered from the onset. The first, as discussed in CHAPTER III, is the problem of not being able to issue a qualification statement in the form of Action X where . . . <COMPONENT><CONDITION><COMPONENT>. This limitation drives the algorithms to successive retrieval-comparison-retrievals. The sequence can be done in two ways. The elements can be either removed one at a time, compared, and these elements which qualify be retrieved, or all of the elements to be used as qualifiers can be removed at once and stored in a stack for use by successive comparison retrieval operations.

For the very general case of an unknown sized table, removing the qualifying elements or rows one at a time, immediately performing the comparison, and immediately performing the output sequence to store the newly generated table would be the method of choice. This method involved changing primary position between two tables to be compared, and performing a retrieval after each position change. The second method is suitable for smaller tables and limited sized data bases since all retrievals can be done at one time and position need therefore only be established twice per function call. Both methods yield equivalent results and either could be used for this project. The second method is elected because it yields more modularity than the first.

Another consideration is that the interface must work directly with the data values, rather than with pointers to values. This is

balanced by the advantage of having the underlying DBMS do all the work of inverting files and keeping track of all maintainance information. The data is essentially physically independent since the relational system accesses it only through the PLI access method. As discussed in CHAPTER III, there is a trade off between data base complexity and redundancy. The "simpler" the relational data base representation is, the more redundancy must be introduced. The less complex schema was chosen for the data base of the design. However, the design programs will function in generally the same way for any hierarchical implementation. This is because the interface has knowledge at all times of which components are assigned to the elements of any table. Changing the underlying schema only involves changing component numbers (as far as the interface is concerned). CHAPTER VI discusses an extension of this nature in more detail.

The general construction of any algebra algorithm is influenced by the problem of not being able to compare components without first taking out one of the components values. As a result there must always be two loops set up for an operation. One loop retrieves the values from one table in an operation, and the second loop does the comparison and retrieval. Such a constraint enables each algorithm to be designed to always terminate. The outer, or retrieval loop retrieves only a finite number of elements, in fact, it normally issues a locate statement and retrieves those rows which were located. The inner, or qualification loop issues as many locate statements as there are

elements or rows retrieved in the first loop. It terminates because unless the PLI functions perform incorrectly, zero or a finite number of elements will always be qualified.

Each of the algorithms presumes the existence of necessary buffers and stacks for retrieval of stored information. Unless they are crucial to understanding a particular process, small details will be omitted. The program limitation of table size of degree 9 is not considered in the algorithms, nor is the table size of 50 rows a factor.

The first algorithm to consider is the process to retrieve a row (Figure 18).

Algorithm GETROW (N)

N = Number of columns to retrieve

1. INDEX = 1
2. GET the next descendant element of the current row.
   If the operation was successful
   Then place the element into a buffer
   Else process an error.
3. INDEX = INDEX + 1
   If INDEX > N
   Then Done
   Else Go To 2

FIGURE 18

GETROW depends on a primary position having been established at the row level of a particular table. GETROW will finish because it cycles exactly N times and will retrieve each element in a row. Retrieval of an entire table is accomplished by positioning at the table and then retrieving each row in succession.

GETTABLE (Figure 19) will retrieve all rows of a table if there are any rows in the table. It will terminate since it cycles in a finite loop controlled by the number of rows qualified in the locate, and since it controls the number of cycles GETROW will make.

Projection of a single column or a group of columns is accomplished by PROJECT (Figure 20).

The algorithm contains two loops, an outer loop on the number of rows in the table and an inner loop on the number of columns to be projected. The algorithm retrieves only the descendants whose names are specified in the array COLLIST. It retrieves the correct rows because position is established at the table whose name is equal to NAME before row retrieval starts and because the outer loop does not allow rows other than those in the table to be retrieved.

Insertion of a table (Figure 21) is accomplished in the reverse of retrieval. Arrays containing the row elements, table description, and the number of rows to be inserted are passed to the insertion routine. Each insertion creates a new subtree at the specified position.

Two loops must be executed in order to insert an entire table because insertion of the column description information within

Algorithm GETTABLE (NAME,N)

NAME = Table name

N = number of columns

1. Establish a primary position at the table whose
   name field = NAME
2. Count the number of rows in the table (LOCATE)
3. If any rows were located in 3
   Then INDEX = 1, Go To 4
   Else Done
4. Establish a primary position at the next row
   If the operation was successful
   Then GETROW (N)
   Else process an error
5. INDEX = INDEX + 1
   If INDEX > Number of rows located
   Then Done
   Else Go To 4

FIGURE 19

Algorithm PROJECT (NAME,COLLIST,NUM)

NAME = table name

COLLIST = Array 1. Number of allowed columns containing the column names to be projected

NUM = Number of columns in COLLIST

K = Position in an output buffer

1. K = 1

   Establish position at Table = Name

   Count the number of rows and place the number in NROWS

   If NROWS = 0

   Then Done

   Else INDEX = 1

2. GET the NEXT row

   J = 1

3. GET the descendant schema whose Name = COLLIST(J)

   If the operation was successful

   Then place the value retrieved into BUFFER (J,K),

      J = J + 1

   Else process an error

4. If J $\leq$ NUM

   Then Go To 3

5. INDEX = INDEX + 1

   If INDEX > NROWS

   Then Done

   Else Go To 2

FIGURE 20

```
                    Algorithm INSERTTABLE
          ( NAME, CNAME, ROW, TYPE, COMP, NROWS, NCOLS )
            NAME = Table name
            ROW = Array to hold input rows
            TYPE = Array of element types
            CNAME = Array of column names
            COMP = Array of component names
            NROWS = Number of rows to be inserted
            NCOLS = Number of columns (elements per row)

1.  Establish position at the last table in the data base
    INSERT the schema for a table with name field = NAME
    If the operation was successful
    Then INDEX = 1, Go To 2
    Else process error
2.  INSERT the schema for a row element
    If the operation was successful
    Then INSERTROW (ROW(1, INDEX), TYPE, COMP, NCOLS)
3.  INDEX = INDEX + 1
    If INDEX > NROWS
    Then INDEX = 1, Go To 4
    Else Go To 2
4.  INSERT the schema for column descriptor with
    description information COLNAME = CNAME(INDEX)
    TYPE = TYPE(INDEX), COMPNO = COMP(INDEX)
    If the operation was unsuccessful
    Then process error
5.  INDEX = INDEX + 1
    If INDEX > NCOLS
    Then Done
    Else Go To 4
```

FIGURE 21

the first loop would mean there would be as many column description groups as rows. Only those rows which fall within the limits (1...NROWS) are inserted, and each row is guaranteed to be inserted in a separate subtree because a subtree parent (ROW) is inserted before each set of row elements are inserted.

Insertion or row elements (Figure 22) is in a somewhat analagous manner to table insertion, except that position is assumed to already have been established at a row entry. Only those elements which fall within the limits [1...NCOLS] in the element array (ROW) are inserted, and these elements are placed into a record of the type specified in the array TYPE.

The elements of a schema to be deleted must be retrieved before a removal can be made, so at least two operations are always necessary for a deletion to be completed. Removing a parent from a hierarchical data base also deletes all of its descendants, so table removal involves only one positioning followed by a remove operation. Removal of a specific ROW requires locating the row, positioning at the row by retrieval and deleting the row repeating group. This removal also deletes all of the constituent elements, since they are the descendants of the row repeating group.

Removing an arbitrary column is more complicated than simply deleting an element repeating group. It is possible that there are entries for the row at levels below the element to be deleted. If there are descendant entries, then deleting a parent will remove all descendants along with the parent repeating group. In order to avoid

Algorithm INSERTROW

(ROW, TYPE, COMP, NCOLS)

ROW = Array 1. Max possible columns of elements to be inserted into this row

TYPE = Array of types for each element

COMP = Component to place the element in

NCOLS = Number of columns in the row

INDEX = 1

1. INSERT schema with

   VALUE = ROW(INDEX)

   TYPE = TYPE(INDEX)

   COMPONENT = COMP(INDEX)

   If the operation was unsuccessful

   Then process error

2. INDEX = INDEX + 1

   If INDEX > NCOLS

   Then Done

   Else Go To 1

FIGURE 22

this problem, all elements which are descendants of a column to be deleted must be moved up in the hierarchy. The algorithm (Figure 23) deletes a specified column by beginning at the column entry to be deleted, one at a time moving descendants up one level in the hierarchy until the end of the row is reached. The last element is then deleted.

Figure 24 is a driver for the algorithm of Figure 23. The algorithm, REMOVE, takes care of the special case of a table with only one column. In this case it is necessary to delete the entire row. The table itself will stay in the data base until an explicit DELETE TABLE command is given by the user.

The outer loop of REMOVE terminates because it is based on the number of rows actually in the table. The inner loop; REMOVE COL, terminates because it successively steps through a finite chain of elements until it reaches the end (step 3), and then terminates. The algorithm correctly deletes one column entry per call because only the last row element is removed from the data base and because values are moved upwards into the element to be removed, maintaining integrity.

Obtaining knowledge of what component is a given component's direct descendant may be accomplished by table look up or it may be accomplished by originally organizing the component numbers in ascending order. A direct descendant would have a fixed constant component difference from a parent.

The algorithm for selection of all rows with one specific value in a column (SELECT) is shown in Figure 25. The idea of SELECT

Algorithm REMOVECOL

(NAME, COLNAME, COMP, TYPE, COLDEX, NCOLS)

NAME = Table name

COLNAME = Name of column to be deleted

COMP = Array of components

TYPE = Array of element types

COLDEX = COMP index of component to be deleted

NCOLS = Number of columns

1. Establish a position at the Table whose name element is NAME and count the number of rows and place in NROWS
   INDEX = 1, COLSAVE = INDEX
2. GET the NEXT row schema
   If the operation was unsuccessful
   Then process error
3. If COMP(COLDEX) is the last element in the row
   Then Go To 4
   Else Go To 5
4. GET the values for COMP(COLDEX)
   REMOVE the repeating group from the data base
   If the removal was successful
   Then Go To 7
   Else process error
5. GET the values of the direct descendant of COMP(COLDEX)
   Place the values into COMP(COLDEX)
   MODIFY the data base
   If the operation was successful
   Then Go To 6
   Else process error
6. COLDEX = COLDEX + 1
   Go To 3
7. INDEX = INDEX + 1
   If INDEX > NROWS
   Then DONE
   Else Go To 2

FIGURE 23

Algorithm REMOVE

(NAME, COLNAME, COMP, TYPE, COLDEX, NCOLS)

NAME = Table Name

COLNAME = Name of column to be deleted

COMP = Component name array

TYPE = Array of Component types

COLDEX = Component index of the column to be deleted

NCOLS = Number of columns in the row

1. If COMP(COLDEX) is the first component in the row
   AND NCOLS = 1
   Then Go To 2
   Else REMOVECOL (NAME, COLNAME, COMP, TYPE, COLDEX, NCOLS)
   Go To 5
2. Establish position at Table = NAME
3. GET the NEXT descendant row schema
   If the last row in the table has just been processed
   Then Go To 5
   Else
   If the last operation was successful
   Then Go To 4
   Else process error
4. REMOVE the row from the data base
   If the operation was successful
   Then Go To 3
   Else process error
5. GET the column descriptor schema with column name field
   equal to COLNAME
   REMOVE the schema from the data base
   If the operation was successful
   Then Done
   Else process error

FIGURE 24

Algorithm SELECT

(NAME, ICOL, ICOND, VALUE)

NAME = Table name

ICOL = Name of column

ICOND = Selection condition

VALUE = Comparison value

1. Set up a new table called NEWTABLE which has the same row types, name, and columns as NAME
2. LOCATE the rows in NAME where ICOL ICOND VALUE
   (that is, the condition ICOND holds between ICOL and VALUE)
3. If any rows were located
   Then INDEX = 1, Go To 4
   Else INSERT a table schema for NEWTABLE with no rows
   Done
4. GET the NEXT row
   If the operation was successful
   Then Go To 5
   Else Process error
5. GETROW(NUMBER OF COLUMNS IN TABLE = NAME)
   Place retrieved information into a buffer
6. INDEX = INDEX + 1
   If INDEX > Number of rows which were LOCATed
   Then INSERTTABLE(NEWTABLE, CNAME, BUFFER, CTYPE, CCOMP,
   Number of rows LOCATEed, Number of columns in NEWTAB
   NEWTABLE), Done
   Else Go To 4

FIGURE 25

is to place into the new table only those rows of the input table which qualify under the parameter condition of COMPONENT (ICOL) ICOND VALUE. The algorithm terminates by retrieving the rows which qualified and inserting the new table into the data base. SELECT retrieves rows with the correct value in them, since only those rows are located by the LOCATE issued at the start of the algorithm.

It is desired in SEQUEL processing to do a COMPOUND SELECT which retrieves rows where column elements may be equal to any of several values in table column.

Since a query always terminates with a table entry placed on the semantic stack, the appearance of a table entry as an argument tells the semantic routines that a COMPOUND SELECT may be necessary. COMPOUND SELECT first places on a stack all elements of the specified column of the table that holds values which will act as qualifying values. It calls a shortened version of SELECT (SSELECT) as many times as there are new stack entries. Qualified Rows are placed in sequential locations in a buffer. The procedure finishes by inserting the new table in the data base. The algorithm is shown in Figure 26.

Restriction of a table is a relational operation which compares two different row elements and saves those row elements in which the elements satisfy the specified conditions. The procedure involves retrieving both elements and comparing them in the interface program. In this procedure it is not productive to use any procedural language routines because none of the routines supplied provides component to component comparison. Since the interface is already

Algorithm COMPOUNDSELECT

(NAME1, NAME2, ICOL1, ICOL2, ICOND)

NAME1 = Table to be selected

NAME2 = Table with selection values

ICOL1 = Column in NAME1 to be qualified

ICOL2 = Column in NAME2 used as a qualifier

1. Find Component numbers for ICOL1, ICOL2
   Set up NEWTABLE with columns, types, and components equal to those in NAME1
2. LOCATE ICOL2 schemas WHERE TABLE = NAME1
3. If any ICOL2 elements were LOCATED in 2
   Then Go To 4
   Else INSERT NEWTABLE with no rows
4. GET all ICOL2 values and place them on a STACK
   INDEX = 1
5. Repeat SSELECT (NAME1, ICOL1, ICOND, STACK(INDEX))
   INDEX = INDEX + 1
   Until INDEX > Top of STACK
6. INSERTTABLE(NEWTABLE, CNAME, BUFFER, TYPE, COMP, NROWS, NCOLS)
   Done

Algorithm SELECT

(NAME, ICOL, ICOND, VALUE)

NAME = table name

ICOL = Column in NAME to be qualified

ICOND = Condition to hold between ICOL and VALUE

VALUE = Qualifying value

1. LOCATE all rows of table = NAME WHERE COMP(ICOL) ICOND VALUE
2. If any rows qualified
   Then Go To 3
   Else Done

3. Repeat GET NEXT row, GETROW(NCOLS in NAME),
   Place result in buffer
   Until the rows LOCATEed in 1 are exhausted
   Done

FIGURE 26

positioned at a given row for possible retrieval it is expedient to retrieve the two elements and compare them in the PLI routine.

The algorithm for RESTRICT is shown in Figure 27. The procedure RESTRICT will terminate since it cycles on the number of rows qualified by the previous call to LOCATE. It will retrieve the desired information, and only that information, because it inserts only those rows which meet the specified condition: Component (ICOL1) ICOND Component (ICOL2).

UNION is used to append all rows from one table to a second table. The two tables must have the same number of rows, and the rows must have elements of similar attributes. The elements are appended exactly as they occur in the row schema, as appropriate columns must be aligned before union. This algorithm is shown in Figure 28. It will terminate because only a finite number of rows are located and inserted. All rows in both tables qualify for insertion in the new table, and the structure of the algorithm guarantees that only those rows in the tables will be qualified in the LOCATES.

Intersection (see Figure 29) is an occasion for all parameters of the PLI Locate subroutine to be used. If the number of columns in two tables to be intersected is the same, then one table is placed into a buffer and its rows sequentially compared to the second table using LOCATE. Rows which are equal (that is all components equal) are placed in the intersection table. The algorithm cycles on the number of rows in the retrieved table. It will produce the

Algorithm RESTRICT

(NAME, ICOL1, ICOL2, ICOND)

NAME = Table name

ICOL1, ICOL2 = Columns between which ICOND should hold

ICOND = Condition

1. Set up NEWTABLE with columns, types, and components of NAME
2. LOCATE rows of NAME and place the number LOCATEed into NROWS
   If no rows were LOCATEed
   Then Go To 8
   Else INDEX = 1, Go To 3
3. GET the NEXT row
   If the operation was successful
   Then Go To 4
   Else Process error
4. GET the row descendant whose component is COMP(ICOL1)
   Place the retrieved value into COMP1
   GET the row descendant whose component is COMP(ICOL2)
   Place the retrieved value into COMP2
   If either operation was unsuccessful then Process error
5. If COMP1 ICOND COMP2
   Then GETROW (NCOLS in NAME)
6. INDEX = INDEX + 1
   If INDEX > NROWS
   Then Go To 7
   Else Go To 3
7. If NROWS > 0
   Then INSERTTABLE(NEWTABLE, CNAME, BUFFER, TYPE, COMP, NROWS, NCOLS)
   Else INSERT only NEWTABLE
   Done

FIGURE 27

Algorithm UNION

(NAME1, NAME2)

NAME1, NAME2 = Table names

1. If the tables NAME1 and NAME2 do not have the same number of columns
   Then Report error
   Else Set up NEWTABLE with column names, types, and components
   of NAME1
2. GETTABLE (NAME1, NCOLS in NAME1, BUFFER(1,1))
   NROWS = Number of rows in NAME1
3. GETTABLE (NAME2, NCOLS in NAME2, BUFFER(1, NROWS + 1))
4. INSERTTABLE (NEWTABLE, CNAME, BUFFER, TYPE, COMP, NROWS in
   NAME1 + NROWS in NAME2, NCOLS)

FIGURE 28

Algorithm INTERSECT
(NAME1, NAME2)
NAME1, NAME2 = Tables to be intersected

1. If NCOLS in NAME1 = NCOLS in NAME2
   Then Report error, Done
   Else Set up a NEWTABLE with description information of NAME1
2. GETTABLE(NAME1, NCOLS in NAME1, SCRBUFFER(1,1)
   NQUAL = 0, NROWS = Number of rows in NAME1
   INDEX = 1
3. LOCATE rows of table NAME2 WHERE
   COMP(1) = SCRBUFFER (1,INDEX) AND
   COMP(2) = SCRBUFFER (2,INDEX) AND
   .
   .
   .
   COMP(NCOLS) = SCRBUFFER(NCOLS, INDEX)
4. If any rows qualified in 3
   Then NQUAL = NQUAL + 1, Transfer SCRBUFFER column INDEX to
   BUFFER column NQUAL
5. INDEX = INDEX + 1
   If INDEX > NROWS
   Then Go To 6
   Else Go To 3
6. INSERTTABLE(NEWTABLE,CNAME, BUFFER, TYPE, COMP, NQUAL, NCOLS)
   Done

FIGURE 29

correct information because the only elements of the second table which qualify under a given call to LOCATE are those in which both tables have equal components.

Set difference is handled similarly to set intersection as the note in Figure 30 indicates. It is essentially a union of two calls to set intersection. Difference will retrieve all values which are in the difference because it performs a membership test for each row of each table versus the rows in the second table of the intersection. Those rows which fail to qualify any rows are included in the difference. The intersection routine must be called twice to insure that all rows of both tables are considered for membership in the difference.

The JOIN of two tables (referred to as the domain and range tables) is shown in Figure 31. JOIN must first retrieve a specified row element from the domain table and compare the value returned to all of the range elements in the range table, and if any rows qualified, then concatenate the two tables. As discussed earlier, JOIN may accomplish this by retrieving one row at a time from the domain table. This involves keeping a counter and repositioning before each retrieval and comparison, or it may retrieve all rows in the domain table and keep a row index and not have the overhead of repositioning.

The division operator is implemented in terms of repeated table intersections (see Figure 32). Values from the divisor table are stored on a stack. The algorithm creates an initial table with those rows whose dividend fields are equal to the divisor field (the

Algorithm DIFFERENCE
(NAME1, NAME2)

NAME1, NAME2 = Table names for difference

1. If NAME1 and NAME2 do not have the same number of columns
   Then Report error, Done
   Else SCRTABLE1 = INTERSECT* (NAME1, NAME2)*
        SCRTABLE2 = INTERSECT* (NAME2, NAME1)*
2. NEWTABLE = UNION (SCRTABLE1, SCRTABLE2)

INTERSECT is the algorithm INTERSECT with the test at step 4 negated

FIGURE 30

Algorithm JOIN

(NAME1, NAME2, ICOL1, ICOL2, ICOND)

NAME1, NAME2 = Tables to be JOINED

ICOL1, ICOL2 = Columns to be compared

ICOND = Condition to hold between columns

1. Set up NEWTABLE with column names, types from NAME1 and NAME2 concatenated
   [Place NAME1 columns in the first (NCOLS in NAME1) components of the component chain
   Place NAME2 columns in the component chain starting at COMP(NCOLS in NAME1 + 1)]
2. LOCATE all rows in NAME1
   NROWS = Number of rows in NAME1
   INDEX = 0, BUFPOS = 0
3. INDEX = INDEX + 1
   If INDEX > NROWS
   Then Go To 10
   Else Establish position at table NAME1
4. GET the row which is INDEX rows from the start of NAME1
   GET the row descendant COMP(ICOL1), Place in COMPARE
   If the operation was unsuccessful Then Process error
5. LOCATE the rows of NAME2 WHERE COMPARE ICOND ICOL2
   If no rows were located Then Go To 3
   Else J = 1, NEWROWS = Number of qualified rows
6. GET the NEXT row which qualified
   GETROW (NCOLS in NAME2)
   If operation was unsuccessful Then Process error
   Else Place values in BUFFER starting at
   BUFFER (NCOLS in NAME1 + 1, BUFPOS + J)
7. J = J + 1
   If J > NEWROWS
   Then Go To 8
   Else Go To 6

8. GET the row which is INDEX rows into NAME1
   GETROW (NCOLS in NAME1)
   Note: Position must be reestablished at NAME1 row INDEX before
   GETROW can be called. The call to GET1 does both at once
   J = 1
9. Transfer the retrieved NAME1 row to BUFFER starting at
   BUFFER(1,BUFPOS + 1)
   J = J + 1
   If J > NEWROWS
   Then BUFPOS = BUFPOS + NEWROWS, Go To 3
   Else Go To 9
10. If BUFPOS = 0
   Then INSERT NEWTABLE with no rows
   Else INSERTTABLE (NEWTABLE,CNAME, BUFFER, COMP, TYPE, NCOLS in
   NAME1 + NCOLS in NAME2, BUFPOS), Done

FIGURE 31

Algorithm DIVISION

(NAME1, ICOL1, ICOL2, NAME2, ICOL3)

NAME1 = Dividend Table

NAME2 = Divisor table

ICOL1 = Result column from NAME1

ICOL2 = Dividend column from NAME1

ICOL3 = Quotient column from NAME 2

1. Set up a NEWTABLE with columns and types ICOL1 and ICOL2

   Use the first two components in the row chain

2. Retrieve all ICOL3 values from table NAME2 and place them on a STACK

   STACKPT = top of STACK

3. NEWTABLE = SELECT (NAME1, ICOL2, = , STACK(STACKPT))

   INDEX = 1

4. SCRTABLE = SELECT(NAME1, ICOL2, = , STACK(INDEX))

5. NEWTABLE = INTERSECT (NEWTABLE, SCRTABLE)

6. INDEX = INDEX + 1

   If INDEX > STACKPT - 1

   Then Done

   Else Go To 4

FIGURE 32

top element of the stack). A loop is executed for the remaining divisor entries. A new table is created whose divident fields are equal to the divisor field at the current stack top. This table is intersected with the initial table, and the table from the intersection is assigned as a new initial table. The loop is executed using INDEX to step through the table.

The last table will be those rows whose QUOTIENT values appear as elements with each of the divisor values.

Removing Duplicate rows is necessary before output or at the end of any statement (Figure 33). The algorithm retrieves all rows of a table. It sequentially compares the retrieved rows to the stored table by locating all rows in the stored table which have elements equal to the corresponding row element of the current buffer row. If more than one row is located, all but one of the rows is deleted. When the last row in the retrieved table has been compared to the stored table all duplicate rows will have been removed.

The parsing principle of this system is to make a rule reduction in the interface whenever one is made by the parser. This is a design choice. It forces the system to do more work than waiting, for example, for an entire string of ANDS, and then performing one qualification. Such an ability is an enhancement but adds more size to an already large program, and it does not add more power to the system.

The SEQUEL subset is shown in Appendix 1. The grammar itself is LALR (1). The structure allows for algebra and SEQUEL

Algorithm REMOVEDUPES

(NAME, FLAG)

NAME = Table name from which to remove duplicate rows

FLAG = Pointer to a buffer if the table is already in the interface, zero otherwise

```
1.  If FLAG = 0
    Then SCRTAB = GETTABLE(NAME,SCRBUF)
    Else SCRBUF = Buffer pointed to by FLAG
2.  LOCATE rows in NAME
    NROWS = Number of rows LOCATEed
    INDEX = 1
3.  LOCATE rows of NAME WHERE
    COMP(1) = SCRBUF(1,INDEX) AND
    COMP(2) = SCRBUF(2,INDEX) AND
                  :
    COMP(NCOLS in NAME) = SCRBUF (NCOLS in NAME, INDEX)
    If more than one row was LOCATEed
    then J = 1, Go To 5
4.  INDEX = INDEX + 1
    If INDEX > NROWS
    Then Done
    Else Go To 3
5.  GET the NEXT row
    If unsuccessful Then Process error
6.  REMOVE the row from the data base
    If unsuccessful Then Process error
7.  J = J + 1
    If J > (Number of rows qualified - 1)
    Then Go To 4
    Else Go To 5
```

FIGURE 33

statements to be alternated, but not intermixed. Every reduction results in an entry on either the operand stack or the operator stack. Every reduction of type <operand> <comp> <operand> results in the operands being removed from the stack, the operator applied to them and the resulting table being put back on the stack.

An algebra statement always terminates with one table on the stack, the result table. A SEQUEL statement always terminates with two entries, the projection list and the result table. If the parser is completely finished with an expression, the specified columns in the list are projected from the table on the top of the stack, otherwise a table is made from the specified columns and the table is placed on the stack in place of the two input entries.

This chapter has discussed the algorithms used in the interface program. All relational operations were implemented in terms of the available hierarchical DBMS functions. The final two chapters are, respectively, an extension to the system, and the conclusion.

CHAPTER VI

AN EXTENSION TO THE SYSTEM

Smith and Smith [19] proposed a data structuring primitive which defines the relational model in terms of a hierarchy of n-ary relations. The structuring primitive takes advantage of both the relational and hierarchical models by building the inter-relational dependencies into the structure of the hierarchy.

A definition in the above structure depends on the user defining key values. Each element of a table is either specified as being a key element or specified as a non-key element. Each element is also listed in every table with which it is associated.

Defining a relational data base in this structured manner allows each relation to be assigned to a level of the hierarchy. To exemplify the structure this concept imparts to the hierarchy, the relational presidential data base is defined in a sample definition language and the resulting structure is graphed.

PRESIDENTS

NAME : KEY OF ELECTIONS-WON : WINNER-NAME

PARTY : NON KEY

HOME-STATE : NON KEY

The structured decomposition yields four levels of tables rather than the one level of tables used by the system designed in this paper. It is immediately obvious that this data organization is suited to the implementation of a non-procedural query processor.

ELECTIONS-WON

YEAR ; KEY OF ELECTIONS-LOST : YEAR

WINNER-NAME : KEY

WINNER-VOTES : NON KEY

ELECTIONS-LOST

YEAR : KEY

LOSER-NAME : KEY OF LOSERS : NAME

LOSER-VOTES : NON KEY

LOSERS

NAME : KEY

PARTY : NON KEY

The structure generated by this sample definition is:

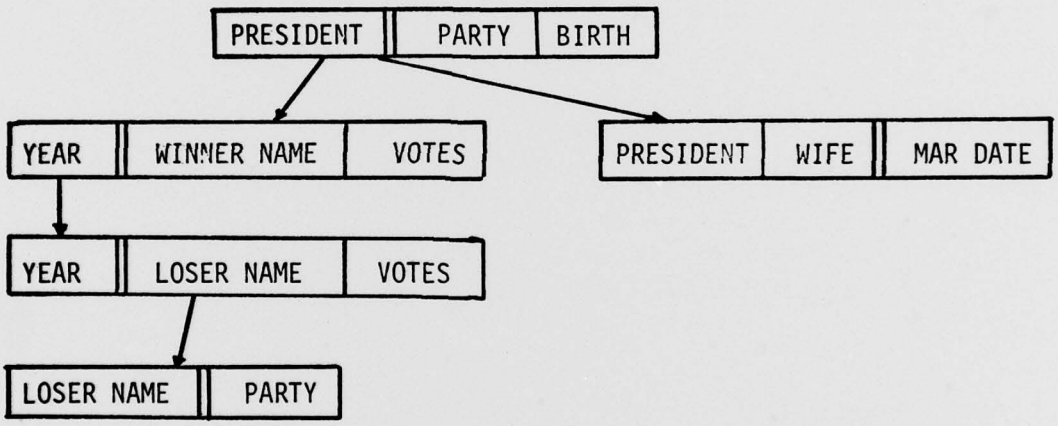


FIGURE 34

Statements such as SELECT LOSERS.PARTY WHERE PRESIDENTS,NAME=ROOSEVELT AND ELECTIONS=LOST.YEAR=1944 AND ELECTIONS=WON.YEAR=1944 ; which are complex in terms of relational algebra operations, may be much simpler in terms of a hierarchical query. The query can be formulated in one statement similar to the example SEQUEL statement.

An algorithm for a query optimizer should be designed which would perform a hierarchical query, a series of relational operations, or a combination of the two methods. This optimizer should make the best use of both models. It should allow comparison of disjoint tables using either the relational algebra or via a schema transformer followed by a hierarchical query.

There are two uses of the work KEY to consider. KEY in terms of the definition language establishes a hierarchical level. An actual management system may or may not regard all elements as elements to be kept in inverted files and by which any row of any relation may be accessed.

If all elements are key elements, then the functioning of the algorithms of CHAPTER IV remains unchanged. Since each algorithm involves retrieval followed by qualification, followed by retrieval of qualified data sets, each algorithm will work properly as long as all elements of a row may be qualified in a query.

If only the elements designated as key are used as actual keys in the hierarchy, a problem arises. Qualifications cannot qualify non key elements, and much of the comparison work must be done by the interface program. For example, in removing duplicate rows from a

relation, all elements need to be available for comparison. The interface must retrieve all elements of a table and compare all non key elements of all rows with the same key. The data base is smaller than it is when all elements are key elements, but algebra operations require longer to finish.

A solution to the above problem is to restrict the relational operations to key values only. This is a reasonable approach since it places the user in the position of deciding the size of the data base as well as the scope of the operations performed in it. Having few key values moves the data base away from algebra operations and towards the non-procedural operations.

In conclusion, an enhancement of the system designed in CHAPTERS III and IV is the conversion of the underlying data base to the general structure proposed by Smith and Smith, the addition of a data base definition program, and the construction of a query optimizer.

CHAPTER VII

SUMMARY

This paper has discussed some of the relationships between a relational data model and a hierarchical data model. It designed a relational interface for implementation as a front end to a hierarchical data base.

Nine choices for structuring the underlying hierarchical data base were discussed. A structure which preserved the table integrity and row integrity of relations was selected.

All relational algebra operations were designed in terms of the hierarchical operations. Two approaches, one applicable to large data bases and the other more suitable for small data bases, were noted to be possible for each operations.

Areas for possible future research were pointed out. An enhancement to the current systems was proposed. This would add a definition module and a query optimizer to the system, making more efficient use of the structure of the underlying hierarchy.

APPENDIX

BACKUS-NAUR (BNF) for the SEQUEL subset:

```
 1    <SLIST> ::= <STATEMENT>
 2              / <SLIST><STATEMENT>

 3    <STATEMENT> ::= <CONTROL>
 4                  / <DEFINE>
 5                  / <ALGEBRA>
 6                  / <QUERY>
 7                  / <DML STATEMENT>

 8    <CONTROL> ::= EXIT ;
 9                / DEBUG ON <DBLIST>
10                / DEBUG OFF <DBLIST>
11                / SHOW ;
12                / DATABASE IS <DBNAM> PASS <PASSWRD> ;

13    <DEFINE> ::= NEW DATABASE IS <DBNAM> PASS <PASSWRD> ;
14               / TABLE IS <DEF TABNAME> <COLDES LIST> : <LITTUPLE LIST> ;
15               / LOAD ;

16    <ALGEBRA> ::= PROJ <PROJ LIST> <ATABNAME> ;
17                / JOIN <ATREF> <RELOP> <ATREF> ;
18                / REST <ATREF> <RELOP> <ATREF> ;
19                / SEL <ATREF> <RELOP> <ATREF> ;
20                / UNION <ATABNAME> <ATABNAME> ;
21                / INT <ATABNAME> <ATABNAME> ;
22                / DIFF <ATABNAME> <ATABNAME> ;
23                / DIV <ATREF> <DIVIDEND> ;

24    <QUERY> ::= <BASIC QUERY>
25              / <BASIC QUERY> ∧ <QUERY>
26              / <BASIC QUERY> ∨ <QUERY>
27              / <BASIC QUERY> - <QUERY>

28    <DML STATEMENT> ::= <DEF TABNAME> = <QUERY>
29                      / <DEF TABNAME> = <ALGEBRA>
30                      / DELETE <DEF TABNAME>

31    <DBLIST> ::= NUMBER
32               / <DBLIST>, NUMBER

33    <DBNAM> ::= NAME

34    <PASSWORD> ::= NAME
```

```
35  <DEF TABNAME> ::= NAME

36  <COLDES LIST> ::= <COLDES>
37                  / <COLDES LIST> <COLDES>

38  <LITTUPLE LIST> ::= <CONSTANT LIST>
39                    / <LITTUPLE LIST> ( <CONSTANT LIST> )

40  <COLDES> ::= <DEF COLNAME><TDES>

41  <DEF COLNAME> ::= NAME

42  <TDES> ::= A
43           / I

44  <CONSTANT LIST> ::= <CONSTANT>
45                    / <CONSTANT LIST> , <CONSTANT>

46  <CONSTANT> ::= NAME
47               / NUMBER

48  <PROJ LIST> ::= <ACOLNAME>
49                / <ACOLNAME> , <PROJ LIST>

50  <ATABNAME> ::= NAME
51               / ( <ALGEBRA> )

52  <ATREF> ::= <ATABNAME> . <ACOLNAME>
53            / <SVALUE>

54  <RELOP> ::= ≠
55            / =
56            / <
57            / ≤
58            / >
59            / ≥

60  <DIVIDEND> ::= <ATREF> <DCOLNAME>

61  <DCOLNAME> ::= , NAME

62  <ACOLNAME> ::= NAME

63  <SVALUE> ::= " NAME "
64             / " NUMBER "

65  <BASIC QUERY> ::= <LABEL> : SELECT <SELCLAUSE LIST> <WHERE CLAUSE> ;
66                  / SELECT <SELCLAUSE LIST> ;
67                  / SELECT <SELCLAUSE LIST> <WHERE CLAUSE> ;
```

```
68   <LABEL> ::= NAME

69   <SELCLAUSE LIST> ::= <SELCLAUSE>
70                      / <SELCLAUSE> : <SELCLAUSE LIST>

71   <WHERE CLAUSE> ::= WHERE <BOOLEAN>

72   <SELCLAUSE> ::= <PROJ TABNAME>
73                 / <PROJ LIST> FROM <PROJ TABNAME>

74   <PROJ TABNAME> ::= NAME

75   <TABLEREF >::= <TABLE NAME> . <COLNAME LIST>

76   <TABLE NAME> ::= NAME

77   <COLNAME LIST> ::= <COLNAME>
78                    / <COLNAME LIST> , <COLNAME>

79   <COLNAME> ::= NAME

80   <BOOLEAN> ::= <PREDICATE>
81               / <PREDICATE> OR <BOOLEAN>
82               / ( <BOOLEAN> )

83   <PREDICATE> ::= <PRED>
84                 / <PREDICATE> AND <PRED>

85   <PRED> ::= <COMPARAND> <RELOP> <COMPARAND>

86   <COMPARAND> ::= <ATOM>
87                 / ( <COMPARAND> )
88                 / <BASIC QUERY>
89                 / ALL <COMPARAND>

90   <ATOM> ::= " NAME "
91            / " NUMBER "
92            / <TABLEREF>
```

REFERENCES

[1] _____. Special Issue on Data Base Management Systems. ACM Computing Surveys, Vol. 8, No. 1, March 1976.

[2] Astrahan, M. et al. "System R: a Relational Approach to Data Base Management". IBM Research Report RJ 1738, February, 1976.

[3] Astrahan, M. and Chamberlain, D. "Implementation of a Structured English Query Language". Communications of the ACM, October, 1975.

[4] Boyce, R. F. et al. "Specifying Queries as Relational Expressions: SQUARE". IBM Technical Report RJ 1291, IBM Research Laboratories, San Jose, CA, October, 1973.

[5] Burger, W. F. "BOBSW - a Parser Generator". SESLTR-7, Computation Center, University of Texas at Austin, TX, December, 1974.

[6] Chamberlain, D. "SEQUEL: a Structured English Query Language". IBM Research Report RJ 1394, May, 1974.

[7] Clemons, E. "Design of a User Interface for a Relational Data Base". Dissertation, University of Pennsylvania, Philadelphia, PA, September, 1976.

[8] Codd, E. F. "A Data Base Sub-language Based on the Relational Calculus". Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, ACM, New York, November, 1971, pp. 35-68.

[9] Date, C. J. "An Introduction to Data Base Systems". Addison-Wesley, Reading Mass., 1975.

[10] DeRemer, Franklin L. "Simple LR(K) Grammars". Communications of the ACM, July, 1971.

[11] Hardgrave, W. T. "Theoretical Aspects of Boolean Operations on Tree Structures and Implications for Generalized Data Management". Computation Center, University of Texas at Austin, TX, August, 1972.

[12] Keel, Tom. "SYSTEM 2000, Version 2.4 User's Guide for UT 2D Implementation". on-line documentation. Computation Center, University of Texas at Austin, TX.

[13] Kent, W. "New Criteria for the Conceptual Model". IBM General Products Division, Palo Alto, CA, unpublished.

[14] Knuth, D. E. "The Art of Computer Programming". Vol. 1 Addison-Wesley, Reading Mass., 1968.

[15] Lowenthal, E. I. "A General Purpose DBMS Kernel". MRI Systems Corp., Austin, TX, to be published.

[16] Martin, James. "Computer Data Base Organization". Prentice Hall Inc., Englewood Cliffs, NJ, 1975.

[17] MRI Systems Corporation. "SYSTEM 2000 General Information Manual". Austin, TX, 1974.

[18] Parsons, Ronald G. "Techniques for Decreasing the Size of SYSTEM 2000 Data Bases". University of Texas CC-TPB-145, June, 1973.

[19] Smith, John and Smith, Diane. "Data Base Abstraction" Computer Science Department, University of Utah, Salt Lake City, Utah, to be published.

[20] Tsichritzis, D. "LSL: a Link and Selector Language". 1976 SIGMOD, ACM, New York.

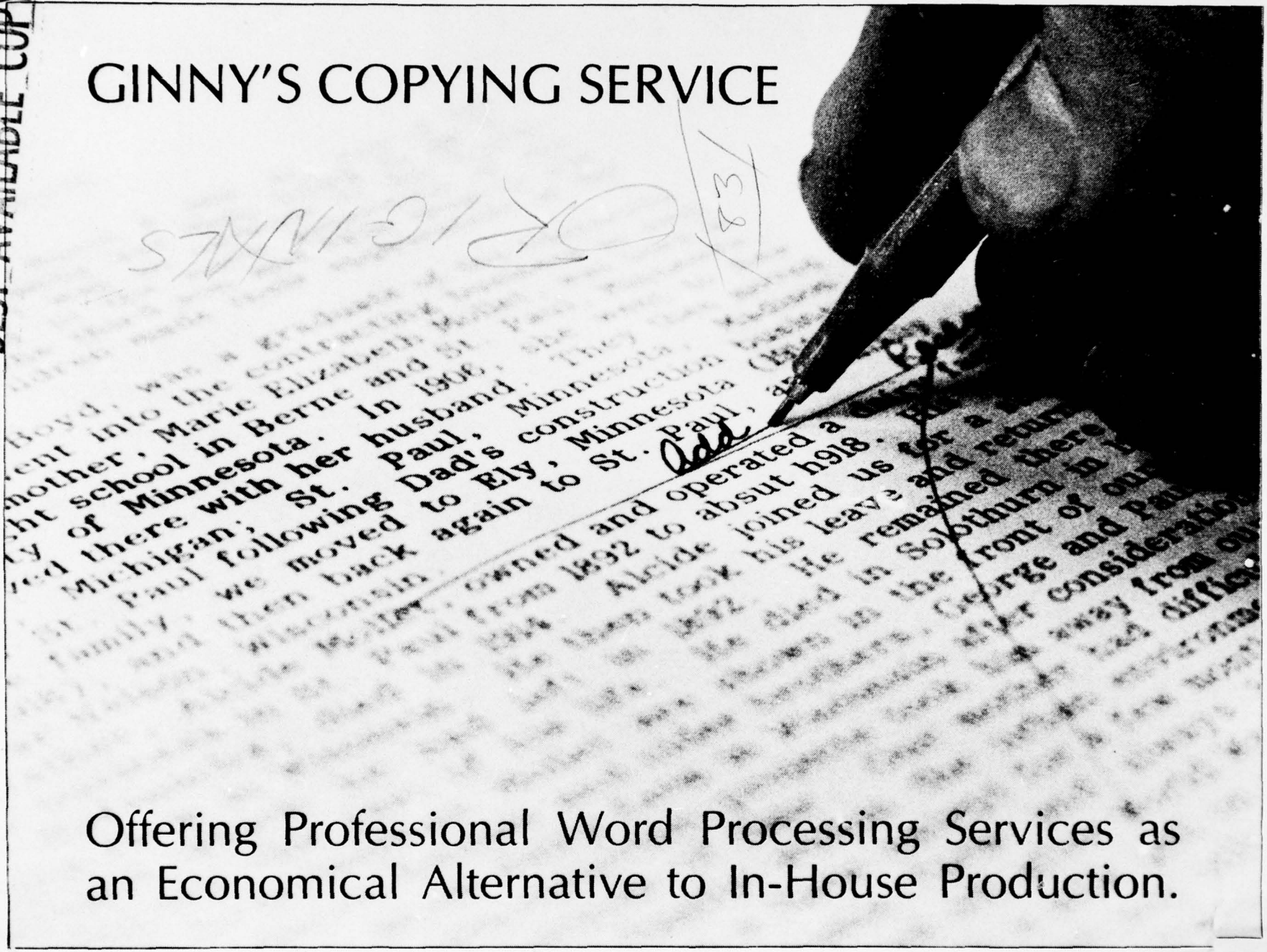
VITA

Ellis Knox Conoley was born in Temple, Texas on March 1, 1947, the son of Maxine Ellis Conoley and Rufus Knox Conoley. He graduated from Lakenheath American High School, Brandon, Suffolk, England in 1965. He completed requirements for the Degree of Bachelor of Arts in Mathematics at the University of Texas at Austin, Texas in August, 1969. On December 22 of 1969 he was commissioned a 2nd Lieutenant in the United States Air Force, and presently holds the rank of Captain. Captain Conoley has served tours of duty in Montana, Southeast Asia, and at the North American Aerospace Defense Command (NORAD) Combat Operations Center at Colorado Springs, Colorado. In May, 1974 he was selected by the Air Force Institute of Technology to pursue a Master of Arts Degree and he entered the Graduate School of the University of Texas in January, 1976.

Permanent address: 10303 Newport Avenue
Austin, Texas

This thesis was typed by Ann M. Patterson.

GINNY'S COPYING SERVICE



Offering Professional Word Processing Services as an Economical Alternative to In-House Production.